A RECONFIGURABLE SUPERSCALAR ARCHITECTURE

THESIS

Christopher B. Mayer
Captain, USAF

AFIT/GCE/ENG/97D-02

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GCE/ENG/97D-02

A RECONFIGURABLE SUPERSCALAR ARCHITECTURE

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Christopher B. Mayer

Captain, USAF

December 1997

AFIT/GCE/ENG/97D-02

A RECONFIGURABLE SUPERSCALAR ARCHITECTURE

THESIS

Christopher B. Mayer
Captain, USAF

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Henry B. Potoczny

Dr. Henry Potoczny

Richard A Raines

Major Richard Raines, Phd.

David M Gallagher

LtCol David Gallagher, Phd.

Preface

This thesis details the design and operation of a reconfigurable superscalar processor utilizing the DLX instruction set and FPGA devices. The thesis is a first part of what will hopefully become a series of thesis efforts here at AFIT. As such, this effort attempts to solve only a small subset of problems surrounding reconfigurable computing. Its chief goal is to outline the requirements for a successful reconfigurable computer, FPGADLX, and to present an initial design and concept of operation. To aid in this research a software simulation of the proposed design has been created. The simulator is capable of running short, uncomplicated programs that allow one to get a feel for the operation and performance of the design.

This effort proved to be extremely frustrating at times, yet came easily at others. From what I understand, most thesis efforts have this kind of "roller coaster" feel to them. Joining me in the front seat for the ride (and often encouraging me to keep going though I felt like getting off) was my thesis advisor, LtCol David Gallagher. His advice, encouragement, and ideas helped me a great deal. Without his hours of mentoring I would not have been able to turn out the kind of product that I have (whether it's good or bad is debatable).

Appreciation goes out to the other members of my committee, Major Richard Raines and Dr. Henry Potoczny, for just watching. Although I didn't interact much with

them because of the nature of my work, I know that they would have been there for me if I had needed them.

Perhaps more than any other person, my wife, Gretchen, was the key to my thesis. By keeping me fed and provided with clean clothes, she allowed me to concentrate more fully on my work and reduced the number of distractions I would have had to endure if on my own. I also thank her for her patience during the late nights and times when things didn't seem to be going well—both during the thesis push and the during normal academic quarters. Condolences are also due for the times she had to tolerate my seemingly mindless rambling about thesis issues she didn't understand. Thanks honey, I love you!

Lastly, I am giving copies of this thesis to my parents Richard Mayer and Sarah Smith. Hopefully, they will see in this a tribute to them for all the hard work, love, patience, and understanding which went into raising me. For without their genes and effort as parents, I wouldn't be completing my master's degree.

Chris Mayer

Table of Contents

List of Figures

Abstract

The invention of the Field Programmable Gate Array (FPGA) has led to a number of interesting developments. One of them is the idea of providing custom hardware support for applications running in a computer. These reconfigurable computers have been shown to dramatically decrease the execution time of a narrow range of applications. Based on past results, attention has subsequently turned to using reconfigurable computing in general-purpose computers (e.g. desktop and workstation environments) for everyday tasks.

This thesis develops a design for just such a computer. Even though the design, named FPGADLX, is based on a hypothetical superscalar processor running the DLX instruction set, it is generic enough in principle to be adapted to any superscalar or VLIW processor on the market today. This thesis begins by examining FPGA technology and reviewing related reconfigurable computing efforts. Based on this review, requirements for a viable general-purpose reconfigurable computer system were developed. In turn, these requirements drove the development of the eventual FPGADLX design. The design includes hardware organization and operation, as well as modification to the operating system and compiler. To better explore the FPGADLX design, a software simulator which can emulate a significant portion of the design and run actual programs has been built. Hopefully, the FPGADLX design as presented herein will serve as a foundation for a generation of superscalar or VLIW reconfigurable computers.

A RECONFIGURABLE SUPERSCALAR COMPUTER

I. Introduction

1.1 Background

Since the dawn of the computer age, computer designers have had to balance application variety with processor speed. In order to run a wide range of programs, one had to use a general-purpose processor design that was high in functionality but lacking in speed. Custom architectures, such as digital signal processors (DSPs) or graphics accelerators, on the other hand, can deliver impressive speed, but only for a limited range of applications. Over the years, device sizes have decreased, allowing engineers to pack more and more computing power into small chips. As gate densities have increased, so have chip speed and features. While this has resulted in impressive speed for modern processors, it has done nothing to change the relationship shown in Figure 1. However, if it were possible to combine the flexibility of general-purpose processors with the performance of custom processors, this relationship could be changed, allowing computers to deliver outstanding

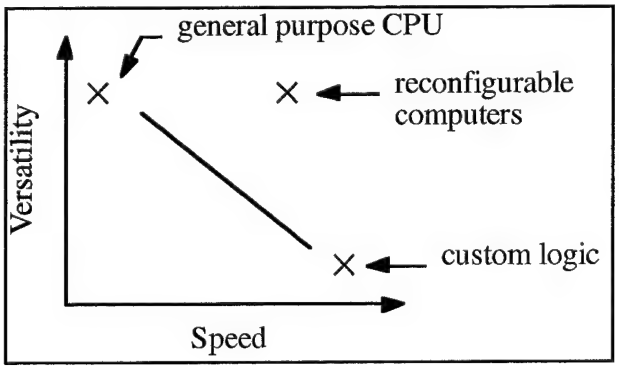

Figure 1. Processor Versatility vs. Speed. General-purpose CPUs can be applied to a wide range of tasks, but sacrifice speed to do so. To gain speed, functionality must be lost, as is the case for customized, application specific logic. Reconfigurable computers hope to break this relationship by providing programmable hardware.

performance across a wide range of applications. Reconfigurable logic may prove to be the cornerstone of this new type architecture called *reconfigurable computers*.

Reconfigurable logic is a term that describes hardware that has the ability to change function. The hardware's purpose is not fixed at fabrication time and the chip must be programmed before it can be used. The number of times a device can be reconfigured ranges from once to an infinity. Reconfigurable devices of the latter type represent a cross between custom logic and general purpose logic. They can be repeatedly tailored to meet the needs different applications while achieving speeds close to that of custom designs. At the present time the favorite type of reconfigurable logic technology, at least as far as reconfigurable computing is concerned, is the Field Programmable Gate Array (FPGA). The FPGA is attractive because it can be repeatedly programmed much like a memory device—in-system and without special devices or uncommon voltages—to perform a myriad of logic functions and algorithms.

A reconfigurable computer (RC) makes use of reconfigurable logic in an attempt to accelerate a broad range of applications. The basic idea is that an RC can provide just the right logic to any application. Whereas a fixed-logic processor must balance the number of functional units it offers programs against the frequency those functional units will be used and the area they consume (which directly impacts cost), an RC can adjust its functions to meet the specific needs of any program. Indeed, simply increasing the number of functional units in a traditional processor will not bring about acceleration "in

proportion to the area these fixed units consume." [17] Instead, it may be more advantageous to add reconfigurable logic to microprocessors. This uncommitted logic would serve as extra logic that could be used by a processor on an application-by-application basis. In contrast to the existing situation in which programs are forced to execute on platforms that may not be ideally suited to them, programs would run on hardware that more closely conforms to their particular requirements. As one might expect, moving away from the fixed nature of current computers towards a malleable environment presents many challenges. On the other hand, it holds the possibility of great increases in performance.

1.2 Problem

Past research dealt with the development of computing systems containing a great deal of reconfigurable logic. These systems were most often created as research tools for a particular problem, or class of problems, especially when an exact solution was not known. By building their own high-performance machines, investigators were spared the expense of developing custom chips or buying a supercomputer. Many of these systems proved to be successful. Unfortunately, they were not adaptable to other types of tasks, partly due to their design, but also because of their large size which made them inefficient for small reconfigurable programs.

More recently, researchers have started to examine reconfigurable computers that have an increased application base. The ultimate goal is to create a reconfigurable

computer for every day, general-purpose use while still achieving decent performance for a wide range reconfigurable applications. One way to do this is to mate a conventional central processing unit with some amount of FPGA logic. The conventional processor core would run all non-reconfigurable applications and execute the software portion of reconfigurable programs, while the FPGA assists by executing some parts of the program in hardware.

Experience with reconfigurable computer systems has shown that they have several shortcomings. Among them are: large system sizes, integrating the CPU and reconfigurable logic, time penalties for configuring the FPGA, writing and compiling programs, timely execution of programs, enforcing correct execution, and communication overhead between CPU and FPGA. Several projects have attacked these problems to varying degrees. Some have even gone as far as suggesting architectures and making sophisticated simulators to verify their designs. While these projects have accomplished their specific goals and produced some novel ideas, it is felt that they have been too narrow in focus and thus missed the larger picture essential to a successful design.

Unlike previous efforts, this thesis takes a broad look at issues surrounding general-purpose reconfigurable computing and proposes an architecture based on those issues. A viable reconfigurable computer will not only have to overcome the problems listed above, but will also have to address other issues. First, it must compete against other improvements to computer designs and show that it is a worthwhile alternative in

terms of cost and performance. Secondly, consideration must be given to how the design will be used. This includes such things as multi-user, multi-tasking environments and the expected application spectrum. Third and last, it must account for impacts to the operation of operating systems and compilers—vital parts of any computer system.

1.3 Objectives

There are two primary objectives for this thesis. The first is to review reconfigurable computing, the technology behind it, and the uses for it, in order to develop requirements for a commercially viable general-purpose reconfigurable computer system. The second objective is to propose a broad design driven by the requirements developed. This design includes not only the hardware and its behavior, but also describes the operation of the system and changes to the operating system and compiler. Since there is more work than can be done in one thesis, the design, while thorough, is intended as a starting point for future research.

There are also several minor goals. Number one is to build a software simulator designed to execute test programs, enabling rough predictions of the design's performance to be made. A second goal is to explore enhancements to the basic reconfigurable computer design. The third and final goal is to suggest directions for follow-on efforts.

II. Background

2.1 Introduction

To properly appreciate the characteristics and promise of reconfigurable computer architectures, a review of previous work and a discussion of current thinking must come first. Presented first is an overview of attributes that make FPGAs a prime platform for reprogrammable systems. Second, a summary of reconfigurable computing devices focusing on a certain class of reprogrammable architectures is presented. Third, the challenges of making reconfigurable computing an everyday reality are presented. Each of the three sections endeavors to not only describe, but also to explore the good and bad points of reconfigurable computing and the technology at its core. The background material provided in this chapter sets the stage for the work discussed in later chapters.

2.2 FPGA Overview

As alluded to earlier, reconfigurable logic is a class of electronic device that can be configured to perform some set of logic functions. One type of programmable logic, the Field Programmable Gate Array (FPGA) is particularly suited for implementing reconfigurable computers (RCs). The following sections provide a quick look at what an FPGA is and why it is an attractive platform for reconfigurable computing.

2.2.1 What is an FPGA?

In general, an FPGA is a collection of logic cells, called logic blocks (LBs), that are interconnected by a network of communication lines. An LB contains some type of logic and usually flip-flops and multiplexers for saving results and selecting results, respectively. A common structure for providing the logic in an LB is an static RAM (SRAM) look-up table (LUT), although other structures such as multiplexers and logic gates are also used. Commercially available FPGAs are composed of hundreds or thousands of LBs. The lines running between LBs are joined together by programmable switches. In most cases the switches are transmission gates where the gate controls the connection of one line to another. A transmission gate's state (open or closed) is controlled by a one-bit register at each gate. Figure 2 shows a basic FPGA design.

Surrounding the LB and interconnect core of the FPGA there is usually a layer of I/O connections that provide the interface between the FPGA and its external pins. The flexible routing nature of the FPGA allows I/O connections to be configured as needed. Sometimes LBs are applied to the task of I/O. Doing this, of course, reduces the device's ability to perform useful logic functions. Typically I/O pins can be established as in, out, or in/out.

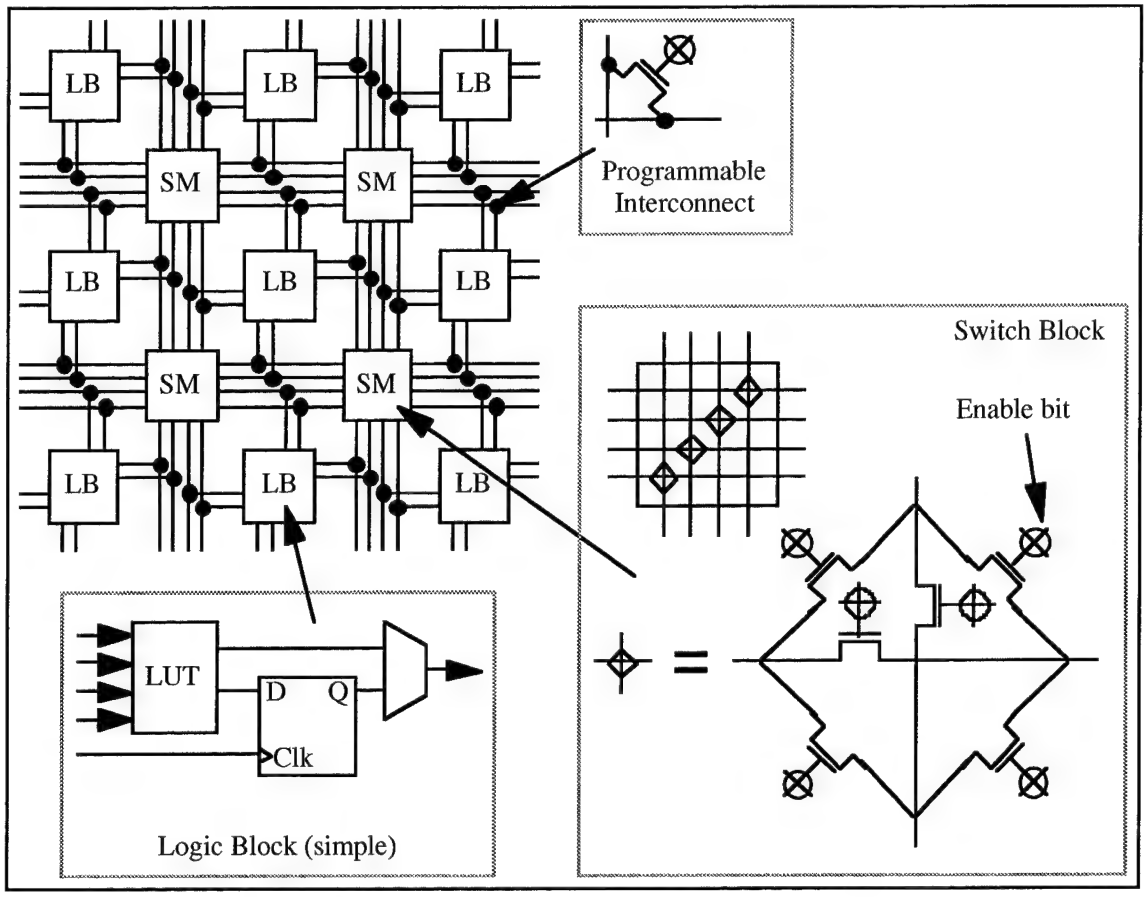

Figure 2. Basic FPGA Structure. The upper left picture shows the regular arrangement of logic blocks with criss-crossing interconnect lines passing through switch matrices. The call-outs show detail for a simple logic block, a programmable switch, and the switch block. Switches are nothing more than n-type transistors. (Drawing based on those found in [45] and [11])

Programming entails changing the FPGA's netlist (sometimes called the configuration memory or bitfile). The bitfile contains the configuration of each LB and the programmable switches. Configuring an LB consists of loading its LUT and enabling other LB features. Once programmed, an FPGA becomes an interconnected network of LBs which is, in turn, connected to the I/O portions of the device. With proper programming, FPGAs become intricately complex logic devices. Because an FPGA is made solely of programmable components (usually SRAM) it can be treated just like any other memory device. In fact, FPGAs can be used as memory in some cases. The

FPGA's memory-like qualities mean that it can be programmed countless numbers of times, providing a different set of logic functions after each reprogramming.

2.2.2 The Good and Bad of FPGAs

While FPGAs might look like a perfect device for reconfigurable computing, there are several down sides to them. The next two sections examine the good and bad qualities of the basic FPGA design as described in 2.2.1. Changes to the basic design are probably needed in order to fit FPGAs into the RC model. Section 2.4.1.2 discusses these changes, but only after reconfigurable computing is investigated in greater detail.

2.2.2.1 The Bad

The basic nature of the FPGA design creates some undesirable side-effects. The next few paragraphs explain what these are and why they occur.

First, FPGAs suffer from long and often unpredictable signal propagation times. The delay is mainly due to the rise and fall times of the transmission gates which provide the connections between routing wires. Designs aimed at alleviating this problem have been proposed [10, 9, 7]. Some chips use longer sections of interconnect that are free of gates in an attempt to shorten propagation delays. While all of these proposals work to some extent, they reduce the FPGA's ability to adapt to diverse applications because they place restrictions on some portion of the design. As will be discussed later, restricting the design may or may not be beneficial for reconfigurable computing.

Further, the speed at which an FPGA can be clocked is highly dependent on several factors. The variable propagation times mentioned in the previous paragraph is one. The number of LB levels from the input of a circuit to the output is another factor. For example, circuits covering a large area of the FPGA and having many stages will usually require a long clock cycle because of compounded propagation and computation delays. However, large circuits are not necessarily slow since a large circuit may only be a few LBs deep and can be clocked very quickly. Careful routing and placement of a function can reduce the effect of delays and thus increase the clock rate. However, the benefit is limited. The latest generation of FPGAs claim clock speeds in excess of 200 MHz. However, the actual operating speeds are much less, more like 50 MHz. Moreover, the actual clock speed is difficult to predict, even in simulation. Often, the clock rate is not known until the design is physically implemented.

Thirdly, FPGAs occupy large amounts of chip area when compared to a typical processor or custom device. The main culprit in this case is FPGA interconnect. The LBs themselves are quite dense. However, to ensure adequate routing of signals between LBs and I/O, large amounts of interconnect must be available. The amount of interconnect can be reduced, but only at the risk of not being able implement some designs due to a lack of routing resources.

The fourth, and final, negative point is that programming is slow. Configuration time is usually measured in the tens of milliseconds. This might seem quite fast in human

terms, but it is slow when compared to clock period of a modern processor which is measured in single nanoseconds. Programming time is a function of the amount of LBs and routing which must be configured. Subsequently, time can be saved by changing only a subset of the FPGA while leaving the rest of the configuration unchanged. FPGAs of this type exist and are commonly referred to as *partially reconfigurable*. Programming speed can also be increased by programming in 8, 16, or 32 bits instead of serially, as was done in older FPGAs. As an example, the Xilinx 4025E, a small to medium sized FPGA (about 25,000 equivalent logic gates), needs 422,128 bits of data for a full configuration. At a maximum clock rate of 10MHz, it takes about 5.3ms to program the 4025E when using an 8 bit bus.

2.2.2.2 The Good

There is no limit to the number of times an SRAM FPGA can be reprogrammed. FPGAs can be continuously reused for widely varying logic functions. In contrast, application specific, fixed-logic devices can only perform one function. Programming an FPGA can be done "in system" without any special voltages or equipment. This is not the case for other programmable devices (like PROMs and PALs) which require high voltage levels and/or extra hardware to program them. Newer FPGAs, such as the Xilinx 6200 family, have been designed to interface with CPUs by allowing the FPGA to be mapped into the system's memory space. [45]

Next, FPGAs provide scaleable logic. That is, FPGAs can be set to perform just the right amount of logic required by an application. A single FPGA can implement simple logic functions as well as more complex ones depending on need. Unlike CPUs which are designed to best handle byte or word length operations, an FPGA can support odd bit-sized operations.

Third, a single FPGA can simultaneously support multiple applications. An FPGA can be partitioned into separate areas each of which can perform its own function. This allows a certain degree of parallelism and multi-tasking. Running multiple jobs increases the demand on I/O resources since a varying number of applications must compete for a fixed amount of I/O. This problem will have to be addressed for RCs where data lines and control lines are probably limited and the reconfigurable logic runs a multitude of functions simultaneously.

Finally, extending the idea of partial reconfiguration, some FPGAs can be *partially reconfigured* while other parts continue to execute normally. FPGAs of this type have been given the name dynamically reconfigurable logic (DRL). [20] DRL helps to hide the cost of programming since the FPGA doesn't stop working on other applications. Current FPGAs with this property include the Xilinx 6200 family. [45]

2.3 Reconfigurable Computing

Reconfigurable computing is a new and emerging field of computer engineering. Terms, methods, and ideas are still being defined. In recent years, several reconfigurable systems have been built. The architectures and capabilities of these machines cover a wide range. The results are promising, but significant problems still exist. To aid in the development of RCs, researchers have developed conceptual systems and proposed classifications for them. This section looks first at the classifications and types of reconfigurable machines. Next comes a review of actual RCs that have an impact on this thesis. Looking at real, working examples exposes some of the pitfalls and advantages of reconfigurable computing. A discussion of lessons learned from these machines concludes the section.

2.3.1 Concepts and Classifications

Reconfigurable computers present a tough classification problem. In general, they can be separated into groups based on either their architecture or by how they load FPGA configurations—i.e. their *configuration method.* To better understand the design and operation of reconfigurable machines, both of these classifications need to be considered. While these two groupings treat RCs as an extension of traditional computing systems, one can also look at traditional systems as a subset of reconfigurable architectures. The following sections examine RC designs from architecture and configuration method viewpoints and then look at rethinking the nature of RCs.

2.3.1.1 Architecture

Independent of the underlying hardware, reconfigurable machines can be split into four categories. The categories are based roughly on how much of the computer is reconfigurable and how much is fixed. Another consideration is the level of sophistication of the logical operations—grain size—the RC can handle. The next four sections explain each division.

2.3.1.1.1 Custom Computing Machines (CCMs)

RCs in which reconfigurable components replace the traditional CPU form this category. The control and logic of a CPU are all implemented with FPGAs and programmable routing chips. A simple example is shown in Figure 3. Two of the best known examples are the University of Toronto's Transmogrifier [14] and Transmogrifier-2 [38] systems. With the number of gates in a CPU running into the millions, CCMs usually require more than one FPGA for implementation. However, this is not always the case as illustrated by [52] and [51]. The clock speed of these types of systems is limited because of their large sizes. Even the single chip systems use most of the available area, which results in long propagation times. Configuration time also is a major concern. The entire system, or at least significant portions of it, must be configured prior to use. Times in the seconds are not unheard of. Combined, configuration time and clock speed severely hinder the performance of CCMs, making it unlikely they will ever be suitable for general use. However, because they are, in effect, real CPUs, CCMs are frequently

used as an alternative to software simulation for testing CPU designs before committing them to silicon. This provides at least one niche role for CCMs. Even though automated design tools exist, human intervention during the construction of these machines is crucial because of their large sizes and level of sophistication.

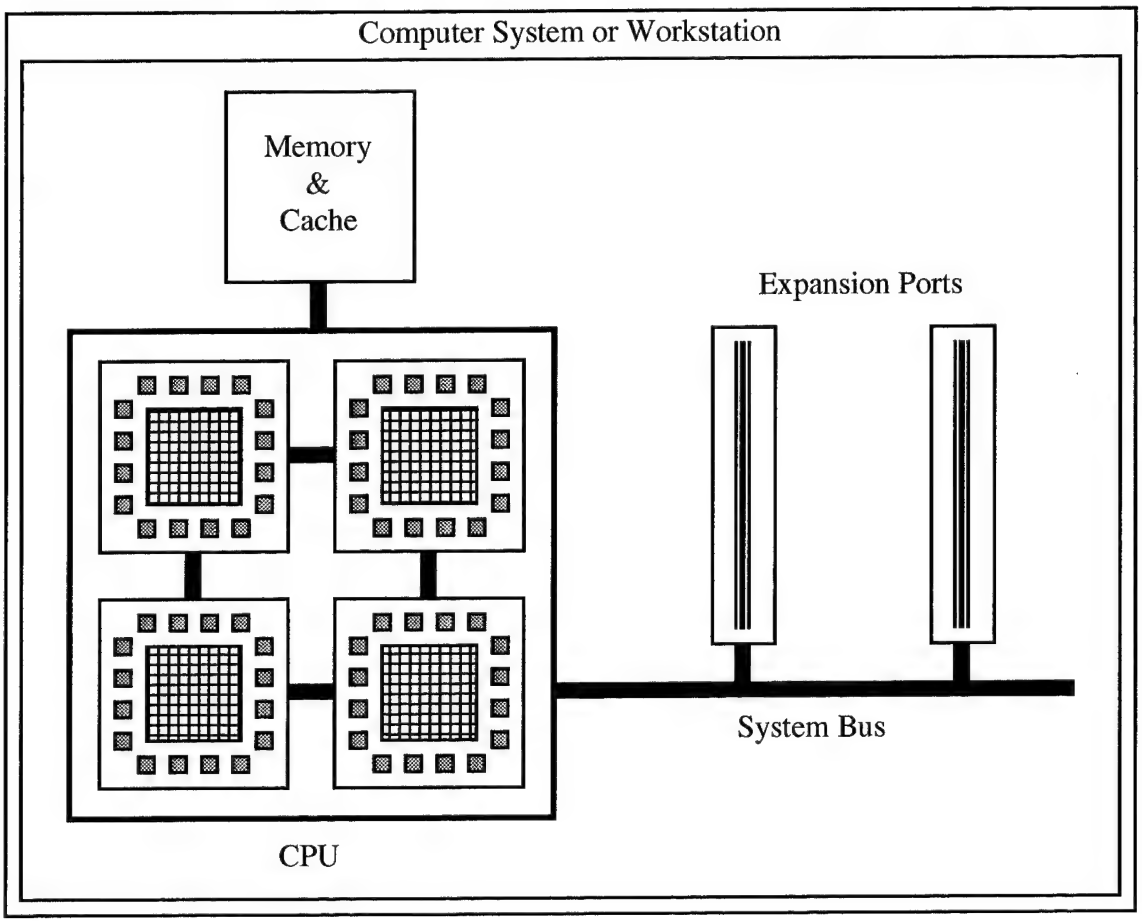

Figure 3. A Custom Computing Machine. A CPU constructed from FPGAs replaces the traditional fixed-logic CPU.

2.3.1.1.2 Loosely Coupled Co-processors (LCCs)

Another approach is to place FPGAs on a card which resides on a computer system's bus. The card acts a co-processor for the system's CPU. The CPU is

responsible for loading the co-processor with its program and data. Once configured, the co-processor acts like any other expansion card. This model is a natural extension of current computer systems in which video cards, DSP boards, and communication boards are already employed in a similar fashion. Figure 4 shows an example LCC. On the low end, the devices on an LCC card usually consist of a few FPGAs and some local memory. More sophisticated LCCs add math processors, routing and communication chips, and RAM to the mix. Often, the high-end machines consist of two or more cards which are connected by their own high-speed bus.

Like CCMs, LCCs have slow clock speeds, long configuration times, and lengthy design times. However, they have been shown to deliver supercomputer-like performance for some applications since their designs and method of operation can be carefully tailored. Commonly, LCCs have been used for large-grain sized problems in which the LCC can perform much of its work without communicating with the host system. The reason for this is that the overhead incurred by exchanging information over the system's bus is quite high. Therefore, to maximize performance, communication must be kept at a minimum.

The machines in this group were among the first viable RCs and still represent the largest collection of FPGA-based machines. Some of the more famous and well-documented examples of this class of reconfigurable machine are the Splash [22] and Splash 2 systems [3,4,42,29,1], PAR-1 [13], and MORRPH [18].

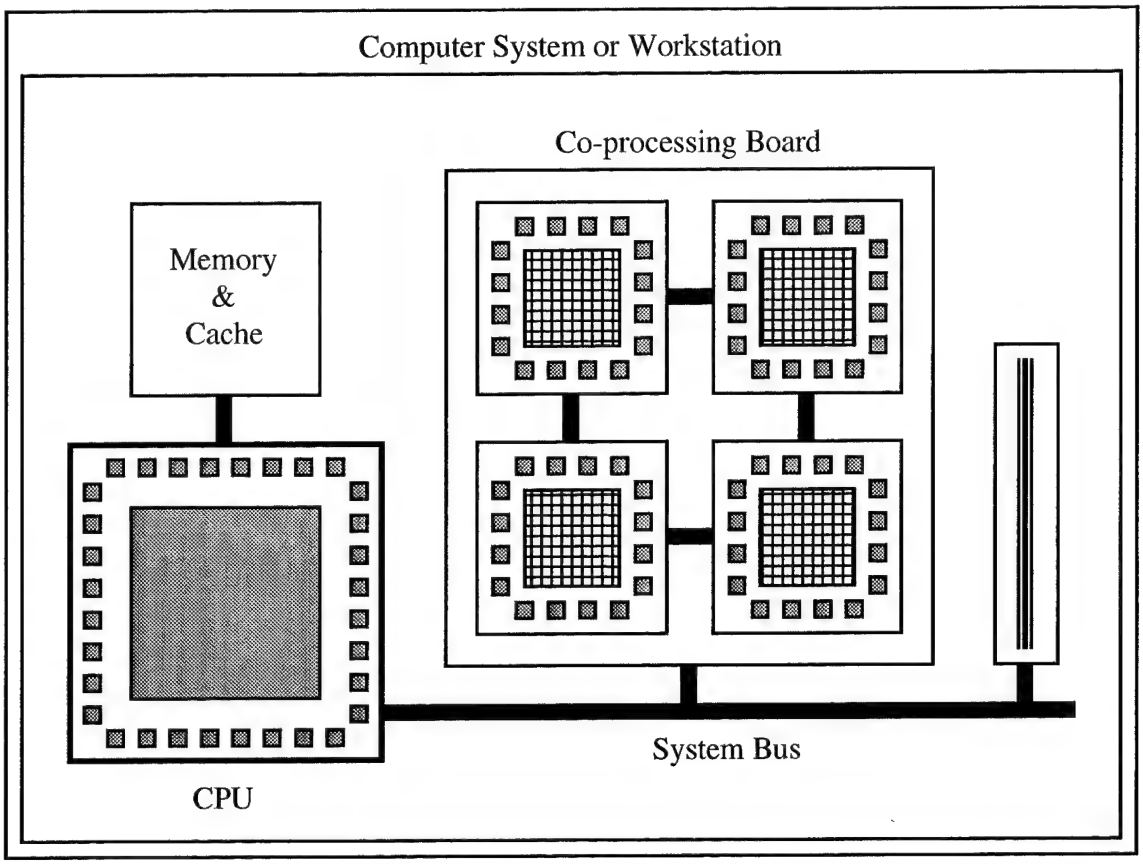

Figure 4. A Loosely Coupled Co-Processor. The FPGAs are placed on a board which resides on the system bus. The co-processor's execution is controlled by the CPU.

2.3.1.1.3 Closely Coupled Co-processors (CCCs)

In order to reduce the communication overhead of the LCC model, the co-processor can be moved closer to the CPU—usually on the processor's local bus as shown in Figure 5. The tighter combination means that small- to medium-grained applications are now cost-effective. The amount of reconfigurable logic can rival that found in the LCCs, but is usually less. The co-processor may have access to system memory. Although the exact location of the co-processor in relation to any caches is debatable, the CCC may also have its own local memory, in effect turning the CCC into a scaled-down version of a LCC. Giving memory access allows for work to proceed semi-

autonomously on complicated tasks, potentially increasing the range of applications for which its reconfigurable logic can be applied and the performance of the entire system. Two of the best known systems of this type are PRISM [6] and PRISM-II [5,2]. Another system that appears to fit this category is now under development at Berkeley. [19,8]

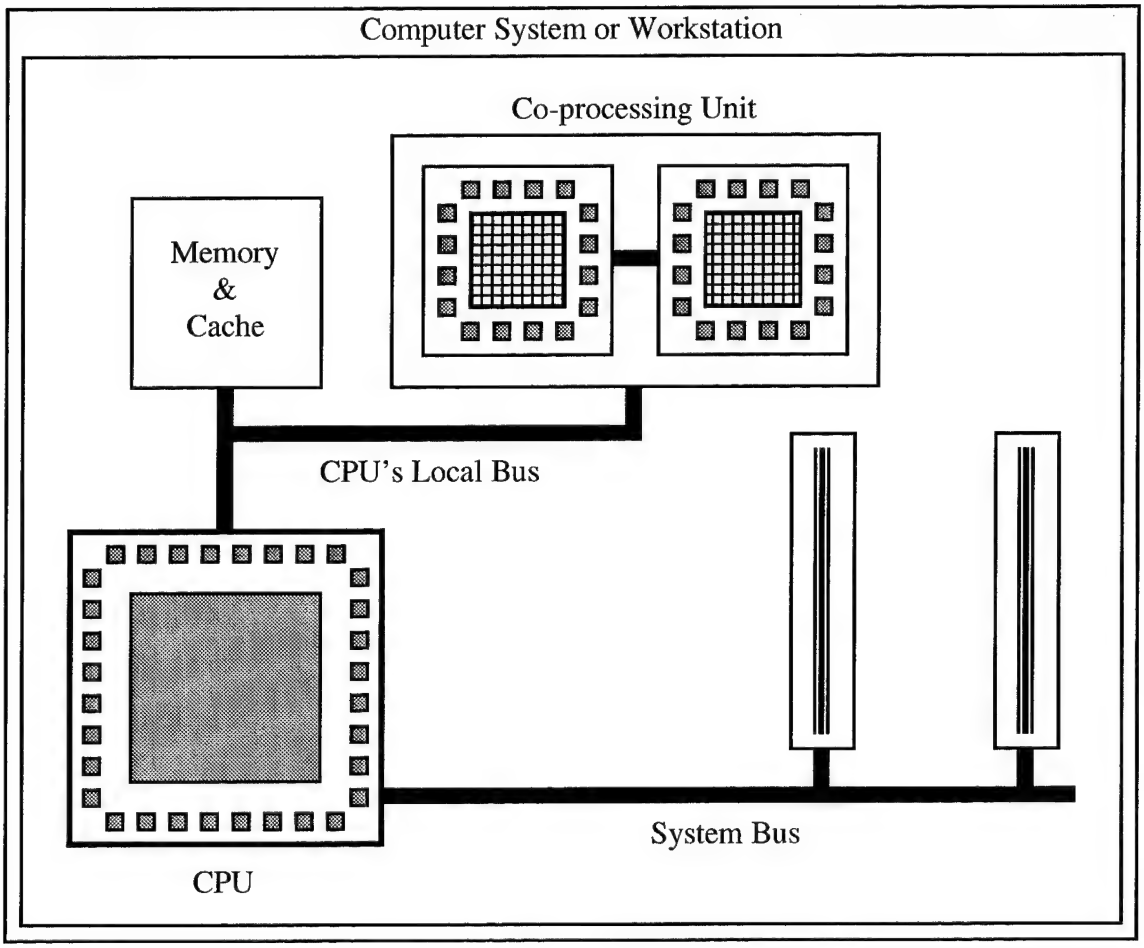

Figure 5. Closely Coupled Co-processor. The FPGA unit is integrated closely with the CPU and may or may not access memory on its own.

2.3.1.1.4 Programmable Functional Units (PFUs)

The last RC model puts reconfigurable components inside the CPU itself (Figure 6). To the CPU, the FPGA resources appear to be just another functional unit, albeit a programmable one, in the execution stage of a processor's pipeline. A PFU system is an extension of the super-scalar type of microprocessor and can be converted for use in VLIW machines. Example machines include OneChip [53], PRISC [44], and Spyder [33]. The prevailing designs include room in the system's instruction set for custom instructions that are defined on a program-to-program basis. The system usually consists of fixed-logic core that provides system control and frequently used instructions and some amount of reconfigurable logic. To provide the most flexible environment for programs, most existing designs allow the number and size of the PFUs to fluctuate. A chief concern in this type of machine is the amount of FPGA resources to add to the fixed CPU core. FPGAs consume more area than fixed logic. Thus, in order to keep the size of a PFU-equipped CPU small, the FPGA portion should be no larger than necessary. However, too little FPGA logic may limit usefulness. A large FPGA might be more flexible, but takes up more space and it may, in turn, be difficult to merge it with the fixed logic due to its unwieldy size.

PFU machines seem to be best suited to small-grained tasks. The reasons for this are many. First, small functions can be executed quickly by the FPGA and therefore may fit the timing requirements of a CPU better than larger functions. Secondly, configuration times are shorter for small function. The CPU will spend less time loading PFU

configurations and more time doing useful computation if the grain size is small. Finally, like normal functional units, the data for a PFU comes from one or two of the CPU's registers at a time. The limited amount of available data restricts the amount of work that can be done. Thus, only a small amount of logic is needed.

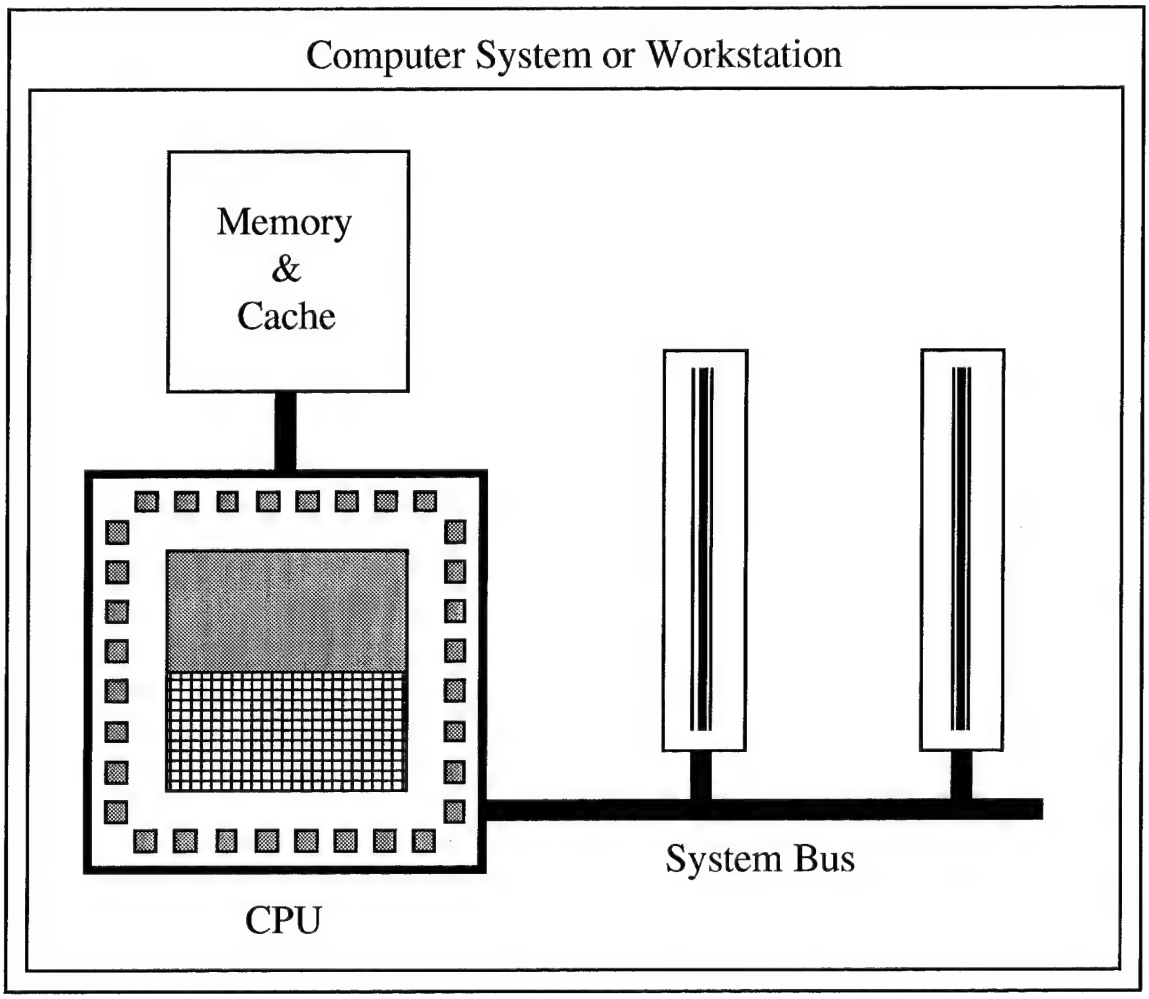

Figure 6. The Programmable Functional Unit Model. FPGA elements are included as part of the CPU and treated as any other functional unit (ALU, FPU, etc.).

2.3.1.2 Configuration Methods

Regardless of physical structure, RCs can be separated into classes by how the FPGA portions are configured and then treated while running an application or task. Machines tend to belong to one of two types as outlined in [32].

2.3.1.2.1 Compile-Time Reconfiguration (CTR)

CTR systems perform one single, system-wide configuration that remains fixed for the life of an application. Here application means a certain hardware image or system-wide configuration that can consist of one or more individual tasks or programs. Everything needed to run an application—logic routing and placement on the FPGA mostly—is determined prior to execution. When an application is to be run, the entire RC stops execution, the new application is loaded, and then execution begins on the new application. The appeal of CTR is its design simplicity; each application is a static object which can be handled separately from other applications. The earliest RC systems all used CTR. CTR is still prevalent amongst CCM and LCC machines because of their typically large and intricate designs.

2.3.1.2.2 Run-Time Reconfiguration (RTR)

If a machine is run-time reconfigurable, then it has the ability to allocate hardware to an application at run-time. The allocation order may be predetermined or not, the key idea is that an application can change its configuration while executing. RTR is akin to the

concept of virtual memory in which an entire program need not be completely in memory in order to execute. Because RTR allows configuration swapping during execution, applications can be larger than the FPGA resources available in an RC. Hence, the FPGA size needed for an application declines, reducing area and system price in the process. For local RTR to be possible, the FPGA must be at least partially reconfigurable. A better platform for local RTR would be dynamically reconfigurable FPGAs, however. With dynamic configuration, productive work can continue in the FPGA while reconfiguration takes place. Local RTR seems to be a key ingredient in building workable, everyday RCs (as later sections will explore).

As one might expect, RTR does not come for free. First, RTR is suitable only for applications that can be divided into time-exclusive segments capable of fitting in the FPGA. This is known as temporal-partitioning. Currently, effective partitioning requires some amount of human involvement, therefore making RTR unattractive to those not wishing to invest the effort. As will be seen later, compiler technology may eliminate this problem. Second, the time penalty paid in configuring the FPGA means that, to overcome the penalty, an application must remain in the FPGA for a reasonable amount of time. This also implies that the application be used frequently while it is loaded. Finally, the results from one configuration might be wiped out when a new configuration is loaded. Some way must be provided so that results can be forwarded from one configuration to another.

RTR can be further broken down into two techniques. The first is global RTR. In this method the entire FGPA is reconfigured any time the FGPA needs to be changed. This is not too different from CTR except that the application doesn't change. The main advantage of global RTR is its simplicity. Each FPGA configuration can be treated as its own entity and can be placed and routed easily by CAD tools. The second technique is called local RTR. In local RTR subsets of the FPGA are configured as needed while the application executes. Figure 7 illustrates this idea. This allows for finer granularity of functions and reduced configuration times compared to global RTR. It also means that all functions don't have to be temporally exclusive since some functions can remain in the FPGA until needed again if the space they occupy is not needed by other functions in the meantime. Accordingly, local RTR can be applied to applications that don't need an entire FPGA or don't divide well temporally. This may result in more efficient use of the FPGA since only the currently active parts of an application have to be loaded at any one time.

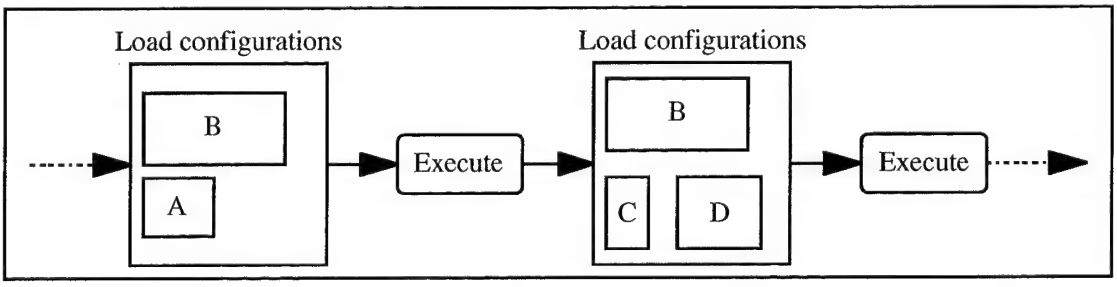

Figure 7. Run-time Reconfiguration. Shown is an FPGA undergoing two reconfigurations during the execution of some program. After each configuration load phase the application executes using the functions loaded on the FPGA.

Unfortunately, the flexibility provided by local RTR increases the complexity of the system. The ever changing functions on the FPGA make it difficult to ensure that functions that may need to communicate with each other can always do so. Something must also guarantee that function footprints don't overlap each other or else the possibility exists that some applications may not run. Meeting these two requirements is tough because the functional mix is not always going to be the same (especially in a multi-programming system) and because current FPGA CAD tools were not built with local RTR in mind.

2.3.1.3 Rethinking the Reconfigurable Computer

Throughout this chapter, RCs have been treated as though they were a remarkable new type of computing system that would drastically change the way computers are built and operate. In some respects this is true. However, one proposal considers conventional processors to be a subset of reconfigurable processors. [23] An ALU is presented as an illustration of this idea. Similar to an FPGA, the ALU can perform different operations based on the reconfiguration data (instruction) given to it. More specifically the ALU is characterized by a dedicated port for instructions, the capacity to perform only a few different operations, and the ability to undergo rapid and frequent reconfigurations (in other words, it can perform a different operation on each clock cycle). The number of bits needed to specify an ALU configuration is small, usually less than ten. This allows less than 2^{10} possible configurations—operations—for the ALU. The ALU provides good, flexible performance for a wide range of applications. In contrast,

reconfigurable logic needs thousands of bits for a configuration and thus has millions of functions.

Looking at computing in this fashion, it appears that the real difference between classical computing and FPGA computing is not in the hardware itself, for the hardware can be constructed so that FPGA components integrate smoothly with fixed-logic. Instead, it is a matter of the number of bits needed to define a function or operation. One of the biggest challenges for reconfigurable computing is managing the large amounts of configuration information.

2.3.2 RC Review

The architectural and classification issues covered thus far make it possible to better understand actual reconfigurable computers. The RCs presented in the following paragraphs have been selected based on their relevance to this thesis. For each RC covered, a quick summary of its architecture and operation is given along with significant conclusions and important ideas stemming from the machine's design.

2.3.2.1 PRISM and PRISM-II

Athanas and Silverman examined problems and proposed solutions involving dynamic instruction set computers. [6] While the main emphasis is on compiler techniques for building code for an FPGA co-processor, the investigation also involved a proof-of-concept system using an 10-MHz Motorola 68010 processor and a system

made primarily of four Xilinx 3090 FPGAs. The hardware design makes this CCC machine with a method of operation resembling global RTR. The compiler and hardware system was named PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis). The PRISM compiler examines C source code, picks high-impact sections of code for placement into reconfigurable hardware, generates the necessary hardware net-lists, and produces an integrated software and hardware executable. Despite an overhead of 48 to 72 clock cycles to compute a solution on an FPGA, the resulting speedups indicate that using reconfigurable hardware make reconfigurable computers worth investigation.

Based on the success of PRISM, a second generation project, PRISM-II, was conducted. [5, 2] PRISM-II addressed some of the problems uncovered with PRISM. Chief among these were a more aggressive compiler that could extract more functions for placement in reconfigurable hardware and more efficient (tighter) communication between the host processor and the reconfigurable hardware. A more abstract goal of PRISM-II was to create a system powerful enough to run real world—not toy or example—programs while being cost-effective for public acceptance. PRISM-II again demonstrated that FPGA-based systems do provide application speedup. Experiments showed that the design and placement of functions in the FPGA makes a difference in performance with payoffs increasing in direct relation to function size and complexity. On the software side, the project expanded the number and types of programming constructs that could benefit by executing in reconfigurable hardware.

2.3.2.2 Dynamic Instruction Set Computer

The Dynamic Instruction Set Computer (DISC) is a system that implements an arbitrary instruction set machine on a single FPGA. [51] DISC is a CCM that uses local RTR with partial reconfiguration. The design uses part of the FPGA for a fixed controller and the rest as space for storing custom instructions. The only built-in instructions are those for execution control and data movement to and from memory (e.g. loads, stores, jumps, etc.). Instructions can occupy an arbitrary amount of FPGA space and are limited only by the physical size of the FPGA minus the fixed controller. To facilitate the placement of instructions and pack instructions as closely together as possible, instructions are laid out horizontally across the width of the FPGA and can be any number of LB rows tall. Dedicated communication lines run vertically through the FPGA connecting instructions to the global controller and each other. Intra-instruction communication uses the FPGA's horizontal interconnect lines. An executing program must load its own instructions as needed for execution. New instructions may load in any available space. When no space is open, old instructions are retired from the FPGA in a FIFO order until the new instruction can be loaded. Execution time can be extremely slow because of the time required to load new functions. While DISC's performance can't be directly compared to other systems, it does demonstrate the FPGA's capability of providing flexible logic in a constant amount of space. It also illustrates that certain constraints may have to be imposed on both the design of functions and FPGAs in order to make FPGA-based computing manageable.

2.3.2.3 Nano Processor (nP)

The nP [52] is another example of a CCM implemented in a single FPGA. Although the physical layout of the nP differs from DISC's, the theory of operation is much the same in that the system consists of a core which has a permanent set of instructions and controls processor execution and a free area for custom instructions. Unlike DISC, however, the nP is not RTR and must be reprogrammed in order to load the needed instructions for an application. This results in an optimized FPGA netlist and processor operation, but limits the numbers of instructions that the processor can have at any one time and, thus, the range of applications that can run on it. nP is perhaps the first RM to see use in the commercial world where (among other places) it is used in National Technologies, Inc. X2 sound card.

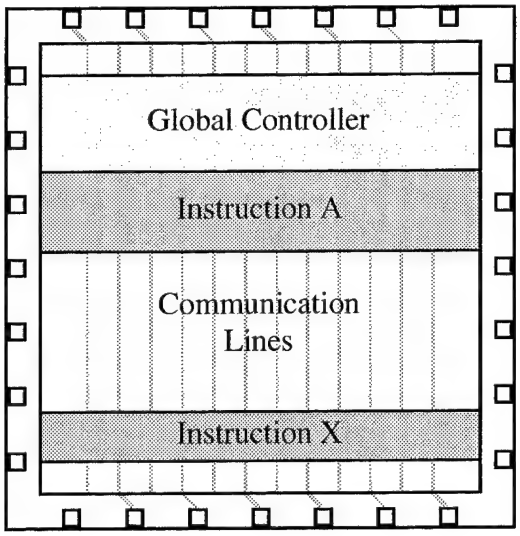

Figure 8. DISC Layout. The single FPGA contains a global controller and space for custom instructions. Vertical communication lines provide connections between the controller and instructions. Horizontal lines (not shown) allow for intra-instruction communication

2.3.2.4 PRISC

The PRogrammable Instruction Set Computers (PRISC) project examines adding programmable functional units (PFUs) to a generic RISC processor. [44] To maintain high clock rates, the PFUs are fine-grained FPGAs capable of implementing a small, fixed amount of logic. Up to 2048 PFUs can be included in the system by storing each PFU

configuration in memory and switching configurations as needed by an instruction. The project describes a processor model, a method of adding PFU instructions to an architecture, and compilation techniques for maximizing PFU utilization. No hardware was built for this effort. Software simulation of PRISC running some of the SPECint92 benchmarks reveals modest performance gains. It must be noted that PRISC relies heavily on compiler transformations in order to create the PFU instructions used in the simulations.

2.3.2.5 OneChip

The OneChip system developed by Ralph Wittig at the University of Toronto falls into the PFU model for reconfigurable computing. [53] OneChip extends the basic MIPS architecture to include PFUs which can implement application specific instructions. Unlike the PRISC project, these PFUs can vary in size and number depending on the application or applications being run on the system. The amount of configurable logic is assumed to be equal to that found on Xilinx XC4010 FPGA—roughly 10,000 logic gates. [45] No hardware was built for OneChip. Rather, it was simulated using the Transmogrifier-1, an FPGA-based CCM capable of emulating complete processor systems. Tests showed respectable performance gains. In one case, OneChip achieved a nearly fifty-fold speedup compared to software routines on a MIPS R4400 for a discrete cosine transform. Three important results came from this effort. First, test applications run on OneChip show that the system overcomes bandwidth limitations of loosely-coupled systems. Secondly, instructions of varying latency were

successfully included into the processor's execution pipeline. Lastly, the MIPS host processor's die area is dwarfed by the size of the FPGA and its associated configuration and state memory as shown in Figure 9.

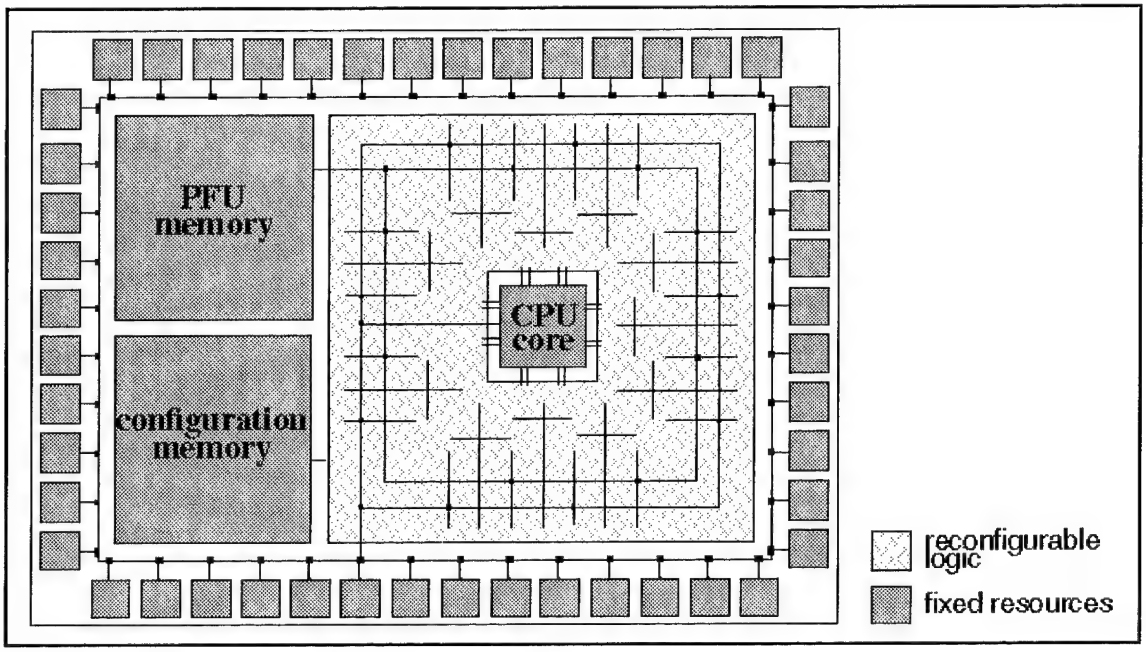

Figure 9. Model for a OneChip Die. The CPU core is dwarfed by the size of the reconfigurable logic and its associated configuration and state memories. [53]

2.3.2.6 Garp

The University of California at Berkeley has embarked on a development effort funded by ARPA to create a chip implementation of a reconfigurable processor. [19] This effort will produce a custom VLSI processor that incorporates PFUs. Deliverables include a compiler that can generate assembly code as well as FPGA net-lists from a high-level language. A recent check on the project's web site [8] shows that their chip, Garp, combines a processor, instruction and data caches, and configurable resources. The

FPGA is a custom design and is capable of being partially reconfigured. Loaded configurations can be quickly swapped in and out of the array. Unlike a pure PFU that is considered part of the processor's datapath, Garp's configurable array has access to the chip's data cache. Although not discussed in the project's literature, one assumes that this is done so that the PFUs can operate somewhat independently of the processor and are thus capable of handling rather advanced computations. Garp seems to be a cross between the PFU and CCC architectures. Another focus of the effort is on creating a compiler for Garp. At the time of this writing, the project is too young to report any results.

2.3.2.7 Chimaera

The Chimaera system [24] has been developed with a focus on extending reconfigurable logic to general-purpose computing. It is another system that integrates reconfigurable logic right into the host processor as PFUs. Architecturally, there are several additions to a processor beyond just the FPGA, as shown in Figure 10. Among them are hardware for partially reconfiguring the FPGA, decoding the PFU instructions added to the host's instruction set, and routing FPGA results to the result bus. The reconfigurable logic is row-oriented, much like DISC, and all rows have read access to the host's registers. This feature allows functions in the FPGA to access a wide swath of data. Additionally, the FPGA has been customized so as to support routing and functions anticipated by the authors (e.g. carry lines for math operations and condition flags). There are two interesting aspects of the architecture. First, there are no state-

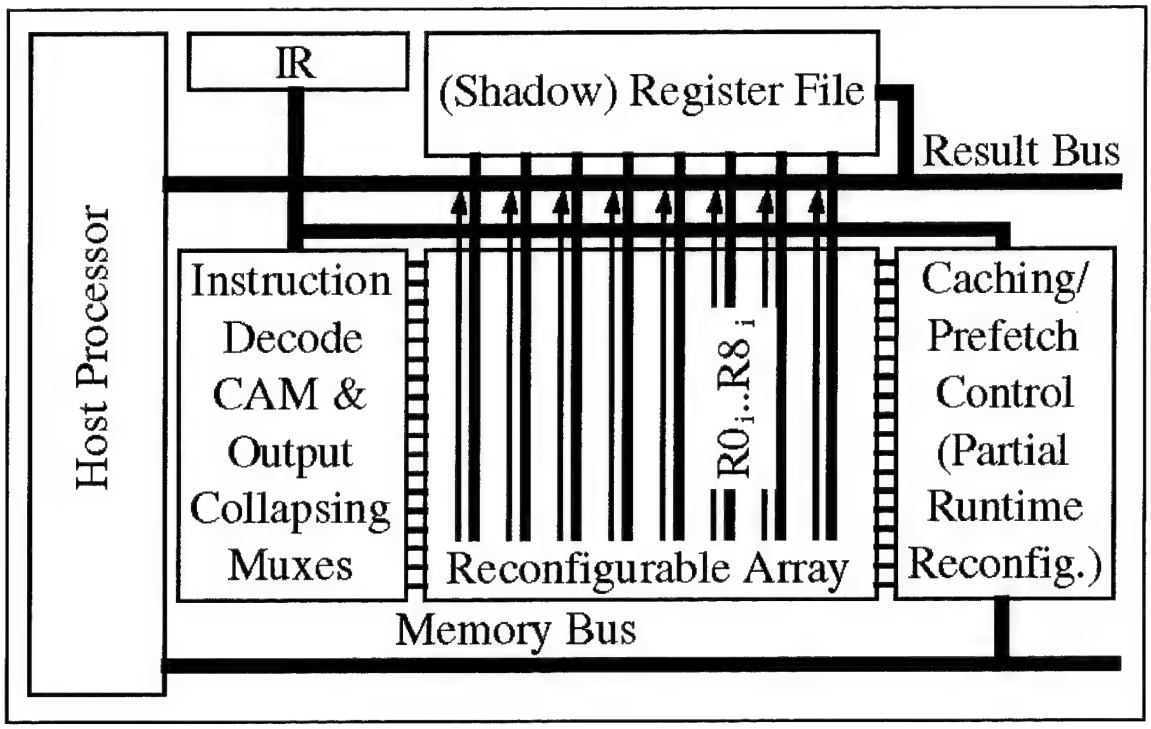

Figure 10. Chimaera Architecture. The reconfigurable hardware is supported by fixed instruction decode and data routing circuits. Other fixed hardware controls the caching, fetching, and partial configuration of the reconfigurable hardware. The reconfigurable array is composed of rows of logic blocks which receive their data from a subset of the host processor's register file. [24]

holding elements or pipeline registers in the FPGA. This means that state-saving on context switches is simplified. Second, the routing structure only moves data downward through the array. Thus, Chimaera's reconfigurable logic is capable of providing only combinatorial logic.

In order to use PFUs, an application calls a special instruction. This instruction contains an instruction ID. If the function is already loaded into the FPGA, the result is immediately written into the destination register specified by the instruction. When the function is not loaded, the system is halted while the function is loaded and resumes when the result is available from the FPGA. Because of the FPGA can only do combinatorial

logic, the contents of the registers providing data to a function must have been prepared and then remain unchanged while the FPGA computes results. Thus, ample compute time must be given before attempting to call a PFU instruction. Chimaera handles this problem by using a compiler to schedule the loading of registers needed by a particular PFU instruction and then executing the instruction only after the result is known to be valid.

Chimaera has been implemented, in simulation, using an RS4000 processor as the host. It has been shown to deliver modest speedups for the Spec benchmark Eqntott and Compress and for Conway's Game of Life when simple optimizations were applied. A detailed optimization of Life produced a speedup of almost 160. While these results are promising, more importantly is that Chimaera is the first complete model of a workable reconfigurable computer designed for general use. Moreover, the model is a natural and relatively simple extension of existing super-scalar processors.

2.4 Lessons Learned and Future Challenges

Numerous examples of RCs abound in addition to those presented in the previous section. While the designs and operating methods vary widely, there are basic issues common to all reconfigurable machines as they exist today. There are additional challenges that need to be overcome if reconfigurable computing is to become a practical reality. These will be covered in section 2.4.1. Meanwhile, this section lists the

problems and design tradeoffs affecting the performance and implementation of current machines so as to give an indication of the challenges ahead.

- **Communication Bandwith and Latency** — Today's machines suffer from a low bandwidth, high latency interface with the host system. The overhead involved reduces the amount of speedup possible and keeps the host and reconfigurable logic from communicating effectively. As a result, the interface between host and reconfigurable logic places constraints on both the use and design of the RC.

- **Application Grain Size** — Due mainly to the interface limitations described in the previous paragraph, current systems are targeted toward a wide range of application grain sizes. Results thus far indicate that grain size corresponds directly to speed. Of course, the application involved has a major impact in determining the achievable grain size.

- **Configuration Time** — Since the time to reconfigure an FPGA is long in terms of the processor clock rate, all systems pay a price for configuration. To be effective, existing systems must counter with infrequent reconfiguration and the use of one configuration for many cycles. This leads to large grained systems in which one configuration is used throughout the lifetime of an application, leaving much of the logic unused during various phases of the application.

- **Floating Point Math** — Experience with FPGAs has shown that they are inefficient at performing floating point operations. Floating point circuits tend to

consume a great deal of chip area and run slowly. Consequently, most reconfigurable systems shy away from performing floating point operations in the FPGA. For this reason, many applications run on RCs tend to have few, if any, floating operations.

- **Application Development** — Designing and implementing programs on current systems are difficult and time consuming tasks handled best by experienced and knowledgeable people. The reason for this is two-fold. First, RC programming involves both hardware and software development. The integration of the hardware and software components in order to create a working system is a complicated issue. Secondly, design and programming tools for RCs are in their infancy. Building an application with today's tools means that the programmer has to do both hardware design (creation of FPGA netlists) and software compilation. Accordingly, the process is fraught with errors and requires the debugging of both hardware and software routines.

2.4.1 Challenges

The development of FPGA-based RCs continues along three main fronts: compilers, hardware design, and operating systems. There are also several ancillary issues which pervade all three fronts and defy easy categorization. All combined, there are several key problems facing RCs that must be overcome. As is often the case, the problems are interdependent such that a change to one often impacts others. This must be kept in mind when finding solutions. Once these problems are solved—there may be

many possible solutions, each with its own merits—practical reconfigurable computing will be very close at hand. This section explores the challenges that face reconfigurable computing in the near future. An attempt has been made to group issues into their own categories, but the high degree of interdependencies makes this difficult.

2.4.1.1 Compilers

Once an RC has been constructed the work is only partly done. For each application run on the machine, a software and reconfigurable hardware executable must be constructed (depending on the characteristics of the machine, the relative sizes of the software and hardware portions of the executable will vary). This is unlike traditional systems in which only an binary executable must be created for each application. For each of the machines reviewed previously—and pretty much for all RCs—the biggest hurdle faced is building the software/hardware image for each application.

As a class, RCs lack sophisticated design and programming tools. On the design side, creating FPGA netlists is an involved task in itself and becomes more so as the size of the circuit increases. Two common approaches for building a netlist are schematic entry tools or hardware definition language (HDL) programs. Schematic entry tools require that the designer manually build the circuit at low levels. While it's possible to create great netlists this way, the process is lengthy and is separate from any software development for an application. HDL tools such as VHDL and Verilog reduce the effort somewhat by allowing the FPGA circuit to be described using something like a high-level

programming language. Neither schematic entry nor HDL provide an easy solution because an application developer must be both a programmer and hardware designer.

Compounding the problem is the fact that the hardware and software for an RC must be developed together for an application. Since some portion of the application runs on fixed hardware and the remaining on configurable hardware, the interaction of the two parts must be tightly integrated in order to run properly. This is true even for CCMs where some portion of the configurable hardware is "fixed" in the sense that it doesn't change from application to application. Developing the hardware and software portions as a whole forces a natural integration of the two parts. Along these lines two main approaches have been considered. The first is to create a new high-level language (HLL) that can deal with both software and hardware. An HLL of this type merges the best of both HDLs and traditional programming languages. The new language keeps the programmer from also being a hard-core hardware designer, but still means that he must learn a new language. Two notable examples of a new HLL for RC programming is Transmogrifier C [21] and the a modified C language for the Spyder [34] project. The second method uses automated tools to compile source code directly into a software/ hardware executable. This method depends on a sophisticated compiler and frees the programmer from much of the hassle of hardware design. Instead the burden is shifted to the author of the compiler. The compiler must create FPGA netlists for each application and blend the resulting hardware and software portions properly. The PRISM compiler is a first attempt at this idea.

One of the biggest problems in compiling a program for an RC is deciding how to split up (partition) the work between the fixed and reconfigurable components of the machine. Hand-built executables enjoy the benefit of human decisions and often have exceptional partitioning and correspondingly good run-time performance. Unfortunately, the effort involved and knowledge required is too much for the average programmer. More advanced tools, like compilers for instance, lack a human's decision-making abilities and may not produce equal results. However, these automated methods are far quicker and easier once the tools are in place. Thus, they are more appealing for large-scale application development.

2.4.1.2 Hardware Issues

2.4.1.2.1 FPGA Layout

Commercially available FPGAs have been designed so as to suit the widest range of uses possible. While this might make sense for the normal FPGA market, it means that FPGAs are too generic and do not fully support the needs of reconfigurable computing. Given that reconfigurable computing becomes a big driver in the computing world, it makes sense that the basic FPGA design change to account for RCs. Placing restrictions on the FPGA design undoubtedly reduces flexibility. However, the anticipated benefits, such as reduced time to place a new configuration, should cover any loss of functionality.

The design must allow for equal access to input and output lines and provide a regular layout and a quantum unit for configurations. Two of the biggest costs of dynamic reconfiguration is 1) the placement of functions on the FPGA so that data can be moved to and from the function and 2) finding space in which to place new functions. Dynamic reconfiguration in this respect is just a variation of the bin packing problem. Any architecture which reduces the complexity of the work involved, increases the chance that a function can be placed and decreases the time needed to place the function. An example of a design that addresses these issues is DISC (section 2.3.2.2). If practical, the architecture could be extended to support datapath operations, like a conventional processor does. This concept is explored in [43]. Since wiring between logic cells consumes a good portion of an FPGA's area, a minimal routing scheme has the added benefit of reducing FPGA size and, thus, cost.

Because the FPGA can contain multiple functions at once, the new design must facilitate the control of functions. That is, the FPGA or some other circuit must control execution. The amount and nature of the control can vary depending on the design of the RC. As an example, if all functions in the FPGA have a predictable execution time (as measured in clock cycles), then a simple controller may suffice instead of a more complex one that might be needed to support functions with irregular execution times. Further, the controller may take an aggressive or passive attitude towards function management. Among the possible responsibilities for a controller are: switching functions off and on, identifying and tracking the physical location of functions, monitoring the state of

functions, stalling the host processor, and, finally, allowing a function to signal completion of a task and then moving results out of the FPGA. The latter problem resembles a superscalar processor's technique of in-order retirement of instructions that were executed out-of-order. Complicating matters, though, is the possibility of the FPGA executing instructions for idle processes.

2.4.1.2.2 Host-FPGA Interface

An alluring FPGA property is its I/O flexibility. Because of the ability to assign pins in a nearly arbitrary manner for the purpose of I/O, FPGAs can be integrated into almost any design. However, placing an FPGA into a system in which its interface is more or less fixed—like a computer—this property is no longer required. More important is that the interface have sufficient bandwidth to support the functions run on the FPGA. Anticipated interfaces include host-FPGA (data and control) and memory-FPGA. Inadequate I/O constrains the usefulness of the reconfigurable array and impacts performance. Too much I/O creates unnecessary overhead and clutters the design.

2.4.1.2.3 Application Grain Size

Reconfigurable computing using FPGAs forces designers to make a difficult compromise regarding application grain sizes for placement in configurable hardware. Simply put, RCs achieve speedup for an application by replacing time-consuming software (assembly instructions) with fast hardware routines executing in the FPGA(s).

The reasons for this are many, but chief among them is ability of hardware to perform a routine, or parts of it, in parallel. This natural parallelism of hardware is already exploited with great success in modern pipelined VLIW and superscalar processors. By converting portions of a software algorithm into a corresponding hardware image, the potential for speedup improves. High speedups usually means using large amounts of FPGA space, which implies lengthy configuration times and a high demand for input data, which in turn requires high a bandwidth connection to the reconfigurable logic. However, using a large functions require that the reconfigurable hardware be placed away from the host where bandwidth and communication latency are lowest. To make things worse, placing and routing a circuit on an FPGA becomes more difficult as circuit sizes increases, especially beyond one chip.

So far, designers have reduced the demand for input data by giving the reconfigurable hardware its own memory or access (through DMA methods, for instance) to the host's memory. This is far from ideal, however, since adding RAM and granting access to memory increases system complexity, size, and cost. Furthermore, having two devices sharing the same memory structure can lead to problems with memory coherence and consistency. Unfortunately, the problem of large FPGA bitfiles cannot be completely overcome, even by giving the reconfigurable array access to memory, since the time to configure an FPGA depends on the size of the bitfile.

The benefits and problems associated with application grain size can be seen in example systems already built. The large LCC systems, executing large-grained algorithms, have high speedups. However, each application they run must be built by hand, more or less, and has a large configuration time. Further, LCCs are often ill-suited to run any other than the large programs they were designed for. Smaller systems, like the CCCs and PFU types discussed earlier, may not deliver the speedups of the LCC systems, but they are more flexible in how they can be used. They also have less configuration overhead and require fewer FPGA resources. Further, the creators of these small-grained systems have succeeded in turning source code into FPGA netlists with minimal human intervention (and sometimes none).

2.4.1.2.4 Instruction Latency and Hiding

Depending somewhat on application type, the bandwidth between the host system and the FPGA must be large and latency small for timely execution. Of course, large bandwidth and low latency is desirable in almost all computing systems, but RCs have a higher demand than regular systems. The reason for this is that not only must data and results be moved to and from the FPGA, but so must configuration data and possibly some control signals. Bandwidth and latency for communication channels in a system differs. Usually intra-chip lines have the greatest bandwidth and lowest latency. As the communication lines move off the chip, bandwidth tends to decrease and latency increase. For RCs then, LCC systems have the longest latency and lowest bandwidth because they must communicate with the host via the system bus. Conditions are better for CCCs

since they usually have a direct, private connection to the host. PFU machines enjoy the lowest latency and highest bandwidths since the reconfigurable components reside with the host on the same chip.

As already mentioned, two drawbacks to reconfigurable logic are configuration time and an execution time that's longer than fixed logic. Both of these work to the detriment of RCs. However, they can be effectively hidden if the RC is designed correctly. In the OneChip system, for instance, it was shown that the execution time of the PFUs could overlap with each other and with fixed FUs. The same should be true of other PFU systems, since the PFU is an extension to the tried-and-true FU. For more loosely coupled systems, techniques such as DMA and memory-mapped devices can hide some of the costs of using FPGAs. [43, 35, 30] Using dynamically reconfigurable FPGAs can help to hide costs during partial reconfiguration.

2.4.1.3 OS Issues

2.4.1.3.1 FPGA Management

Merging the ideas of RTR, dynamic reconfiguration, and "hardware caching" can reduce the amount of FPGA hardware needed in an RC. [28] Understanding that memory is denser than an FPGA, the opportunity arises to replace some of the area used by the FPGA with memory—with the resulting area being smaller than the FPGA alone—and still be able to implement as many functions. A hardware cache in many ways resembles

the familiar memory caches found in computing systems. Where memory caching involves the storing of instructions or data, a hardware cache stores configuration data for an FPGA. Because the hardware cache sits close to and is designed for the FPGA, configurations can be quickly swapped into and out of the FPGA. RTR and hardware caching effectively folds many FPGA devices into one, thus saving space and cost. One example of an FPGA design incorporating RTR and reconfigurable logic is the DPGA chip. [17, 15, 16]

The most pressing problem in this area for an OS is the administration of reconfigurable hardware. Similar to the way in which an OS monitors the status of processes and manages memory, a future OS must also be responsible for managing the reconfigurable hardware. The OS will have to associate hardware executables with their parent processes and ensure that the execution state is saved. Akin to paging, the OS will have to decide when to load and remove hardware configurations into or out of the FPGA. This includes deciding not only who should be removed, but also where to load the new configuration in the FPGA so that it has room and resources to operate. It might mean that the OS has to perform limited placement and routing of the new function at runtime. Fortunately, the OS's job can be greatly eased by the hardware. The idea of hardware caching from the previous paragraph provides a fast backing store for hardware netlists. A regular, predictable FPGA design developed for this sort of configuration swapping will almost certainly benefit the OS.

2.4.1.3.2 Multi-tasking Environments

Almost all of today's operating systems feature the multi-tasking of programs which can themselves be multi-threaded. Reconfigurable computing throws a wrench in the works, since operating systems must manage programs comprised of both software and hardware portions. It is unlikely that the computer community would voluntarily give up the power provided by a modern OS. Therefore, RCs should support multi-tasking and multi-threading. Undoubtedly, there will have to be some changes to an OS to migrate it from a fixed-logic system to a reconfigurable one. With careful design, however, the impact can be minimized. Dynamically reprogrammable FPGAs and rapid reconfiguration will greatly enhance the prospects for multi-tasking.

III. RC Justification and Design Considerations

3.1 Introduction

Chapter 2 covered the background of reconfigurable computing by setting basic definitions, examining the work to date, and covering the challenges facing RC designers in the future. One realization from Chapter 2 is that all the previous work in building RCs have been of limited focus. In most cases the designers have built machines or have proposed architectures that, while worthy in their own respects, don't capture the full scope of designing and building a general-purpose RC. Instead, they have focused on individual pieces of the entire puzzle. For example, the two PRISM efforts concentrated mainly on compilation issues, while DISC explored architecture and operating issues. The closest examples of a full architecture and operating description so far are the Garp and Chimaera projects.

With this in mind, this chapter presents the background material needed to arrive at a viable RC design suitable for general-purpose use. First is a look at the need for general-purpose RCs. This includes examining closely the traits of applications expected to benefit from reconfigurable computing. This is a logical predecessor to the second section of the chapter in which the requirements for a RC are developed.

3.2 The Need and Opportunity for Reconfigurable Computing

The ultimate goal of many in the reconfigurable computing field is to make reconfigurable components a normal part of all computers. While this goal seems acceptable on the surface, it begs two very important questions. First of all, is it worthwhile to build RCs? And if it is, are there better alternatives? To answer the first question, one must investigate the need for RCs. This means examining the types of applications that would potentially benefit by running on an FPGA-based RC. If the application base seems broad enough, then RCs should at least be considered as a way to increase the power and performance of general computing systems. To answer question number two, one needs to consider other enhancements to computing systems and compare them to FPGA-augmented systems. The next two sub-sections explore each question in detail.

3.2.1 A Question of Need

It is commonly agreed that one of the principal driving forces behind any computer design is processing speed. Designers are interested in increasing the performance for as many of the anticipated applications as possible. "Anticipated" is a key word, since computer designs, though usually quite similar in nature, are often targeted for different uses. For example, workstations usually emphasize floating point performance while PCs do not. This is because workstations are more likely to be used a scientific or engineering tools than is a PC. Therefore, when developing new computer

designs or enhancing existing ones, designers must constantly evaluate the value of changes and additions to a system with regards to how they affect the performance of applications, or a sub-set of applications, expected to be run on the computer. The case is no different for RCs.

This evaluation can be done in two steps. The first is to investigate the range of applications expected to benefit from reconfigurable computing. This goes beyond whether or not RCs are beneficial, for there are already numerous examples of configurable computers delivering outstanding performance for some select applications. What should really be asked is, "Are there enough situations in which reconfigurable computing could be applied and produce beneficial results?" To date, most current RCs have been applied to niche applications. The second step in this kind of analysis involves evaluating the opportunities available for employing RCs effectively on anticipated programs.

3.2.1.1 Application Range and Characteristics

In examining of the need for RCs it makes sense to compile a list (given below) of applications known to benefit from execution in an FPGA. FPGAs, by their very nature, are more suited to some kinds of computing tasks than to others. The review of RCs in Chapter 2 revealed some of them. The list is by no means complete, but at least serves as a starting point for evaluation. The material in this list come from [5], [19], and [25].

From this list can be extracted the characteristics of programs or algorithms that cause them to be suited for FPGA execution.

1. Low precision digital signal processing. This includes a broad range of algorithms including such things as fast fourier transforms and discrete cosine transforms. Masking and filtering of data can also be considered.
2. Cryptography. Including encryption, decryption, and authentication algorithms.
3. Pattern matching. Includes such things as text searching and text, sound, and image comparison.
4. Data compression and decompression.
5. Routing and path planning: PCB/VLSI routing and terrain mapping, for instance.
6. Low-level protocol management. Examples are TCP/IP and ATM monitoring.
7. Glue logic. On the fly conversion or manipulation of data between two devices.
8. Data conversion: For example, changing formats for digitally stored images.
9. Sorting.
10. Image processing and manipulation. Edge detection, embossing, hatching, and image filters fall into this category.
11. Physical event simulation. Examples include discretized grid problems and systems of linear equations.

12. Mathematics. Long multiplication, modular multiplication, Monte Carlo algorithms, Laplace equation solving, and matrix operations are included.

Careful examination and consideration of the applications in this list reveal some distinguishing characteristics about them. First of all, for most of the applications the majority of the computation is contained in a small, tight kernel of code. Optimizing this kernel results in dramatic improvements in execution time. FPGAs stand a good chance at successfully implementing algorithms that have a kernel of code when the two conditions are met: 1) the small kernel size fits easily into an FPGA, and 2) the amount of use overcomes the load (configuration) penalty.

Second, the kernels can benefit from pipelined or parallel execution. This may apply in varying degrees to each application, however clever partitioning of a kernel can usually produce one or the other, or both. Parallelism can be successfully exploited by the FPGA because the regular array of LBs in an FPGA can be configured to match that offered by the code. Pipelining operations through FPGA functions can overcome the sequential nature of the assembly instructions the functions replace.

Third, the data to be worked on is usually stored close together and is accessed in a regular, predictable pattern. This is good for two reasons. One, the data can be steadily fed into the FPGA so as to maximize the use of the hardware. The expense of configuring the hardware means that the hardware should be used as much as possible. Two, the logic involved in fetching data is kept to a minimum. The FPGA area used for an algorithm is

best spent when focused on the computational portion of the algorithm. Any extra logic, such as an if-then-else structure to determine the next address from which to fetch data, can be potentially wasteful in terms of both chip space and time. (This does not rule out if-then-else or other reaction type logic functions from being implemented in an FPGA, since FPGAs are quite capable of evaluating logic and making decisions.) Finally, many of the algorithms can be implemented as systolic systems. That is the algorithm can be combined with hardware to form a simple, regular compute structure consisting of interconnected cells that operate in parallel. Systolic systems are regularly employed for signal filtering, pattern matching, correlation, interpolation, polynomial evaluation, matrix operations, and polynomial multiplication and division. [37:491-505] Many cryptography algorithms also lend themselves to systolic machines.

Fourth, these applications involve mostly integer data and operations. Although they can also come in floating-point varieties, a great deal can be done without any floating-point operations at all. That the applications have this quality is not unexpected since floating-point math is one thing that FPGAs don't do well.

Fifth, and finally, some of the applications often feature odd-sized (i.e. not 32-bit) data. This is particularly the case with image processing, data compression, and cryptography which frequently operate on 8- or 16-bit values. FPGAs offer an advantage over fixed functional units because the FPGA can be tailored. For instance, an 8-bit integer multiply is simpler to implement and takes less time than a 32-bit multiply.

Using functional units, which always do a 32-bit multiply, is wasteful since the 8-bit multiply will resolve faster than the 32-bit multiply. However, an FPGA can be programmed specifically for an 8-bit multiply. Depending on the speed of the functional unit and the FPGA, it might be better to do an 8-bit multiply in the FPGA. Being able to tailor FPGA functions to the data also offers the opportunity to pack data for better execution efficiency. Continuing with the multiply example helps to illustrate this idea. A 32-bit functional unit requires 32 bits of data even though it is only operating on eight. However, two 8-bit multiplies can actually be done in the space of 32 bits—multiplying two 8-bit numbers produces a 16-bit result, and two results is 32 bits—and is possible in the FPGA.

3.2.1.2 Opportunities

Now that the types of applications suitable for execution in an FPGA and their characteristics have been covered, we can examine how frequently and in what environments these applications and characteristics appear. It certainly seems that reconfigurable computing has uses in more than one area. Numerous though the opportunities are, they seem to turn up most frequently in a few isolated areas, such as the sciences, engineering, and graphic design. These disciplines already put a premium on computer performance, usually employing powerful workstations for their work. It would appear, then, that RCs might find an immediate use in the areas where workstations are currently found.

However, far more common applications such as word processing, spreadsheets, money managers, games, and various simulations (e.g. Spice)—in short, programs found on most home computers or at the average office—probably won't see much benefit from using FPGAs, at least in the near future. The reason why can be explained easily. To begin with, the diverse nature of the programs run on these computers diminishes the likelihood that any program will exhibit the characteristics which make RCs attractive. When an application is suited to reconfigurable computing, it is likely that only a small and infrequently used portion, such as a find and replace feature of a word processor, will be suitable. Because of this, the opportunity to accelerate these kinds of applications is greatly reduced and it appears doubtful that RCs can be used effectively in the home or office environment.

So, outside of some limited fields of use, it seems that there is little or no way for RCs to be of use. It would help the RC cause if there was a way make reconfigurable computing attractive to a wider range of applications. As it turns out, this may be possible. Research has already shown that clusters of instructions otherwise unfit individually for an FPGA and demanding multiple clock cycles to complete on a traditional CPU, can sometimes be condensed into a few, or one, FPGA-based functions which require fewer clock cycles. [44] It stands to reason that the resulting speedup from this kind of enhancement is probably not going to be as great as placing compact kernel in an FPGA. This is because the instructions are not as tightly packed and that the opportunity to use these condensed instructions may not be uniformly distributed.

However, such a scheme could open up reconfigurable computing to applications that otherwise wouldn't be suitable.

Another factor in determining the suitability of RCs is deciding when to use an application specific integrated circuit (ASIC) instead of relying on an FPGA. FPGAs can provide a custom logic solution to tasks that are so infrequent as to not warrant the expense of a masked-logic circuit. Further, with an FPGA the logic can be changed. This means that many tasks can run in one piece of hardware, instead of having hardware for each application. On the other hand, ASICs are faster than FPGAs and usually require less space for equivalent circuits. ASIC-based computers are easier to program than reconfigurable systems simply because the ASIC system has a fixed design. However, because ASICs are targeted to a specific task, they are inflexible and remain idle when not in use. Consequently, ASICs are attractive as solutions for tasks occurring frequently across many applications—the idle time of the ASIC is distributed among several programs. It is important to realize that FPGAs must compete against other types of devices. Therefore, designers must weigh the merits of the ASICs and FPGA—and perhaps other logic devices or methods to accelerate program execution.

Even though FPGAs are in competition against other technology, recent trends in computer design and utilization might favor them. Over the past several decades the roles which classes of computing systems play have become increasingly blurred. For example, not to long ago mainframe and mini-computers were the dominant scientific and

engineering computing platforms. Today, workstations have replaced them. The latest shift in computer evolution is the merging together of the PC and workstation. Increasingly, PCs are used as low-end workstations while workstations run more and more programs once only found on PCs. Soon the differences between PCs and workstations might disappear altogether. A reason this can, and is, happening is that computers are becoming better at handling a broad range of tasks than they once were. As a result, a single computer can now be expected to serve as a word processor, CAD station, graphic design tool, game machine, and a VLSI circuit simulator. While special boards and accelerators are still used when needed, computers are becoming increasing flexible in the types of programs they can run effectively.

This is where RCs can play a part. The true power of the RC comes from its ability to adapt to changing needs while delivering respectable performance for each function performed. Thus, RCs can further increase a computer's ability to perform admirably no matter what application it runs. As a side benefit, the need for expensive accelerator boards, each of which are only good for a narrow range of applications, may be reduced in RCs since they are, in effect, their own accelerator.

3.2.2 Alternatives to RCs

RCs must not only fight for a bigger application base, they must also compete against other hardware options that increase performance. More specifically, the addition of FPGA resources to a computer system will certainly increase the chip area of the

system. The growth in size is expected to be quite large, given the density of the typical FPGA. But, what if, instead of adding FPGA elements, the extra real estate is used for alternate enhancements such as more functional units, an increased number of registers, or extra amounts of on-chip cache? Certainly, there would be some advantages, although pin-pointing exactly how dramatic is tough.

Adding registers and cache would have some benefits. However, a problem with both is that they have diminishing returns. For example, doubling the number of registers from 16 to 32 may result in tremendous gains in performance, but going from to 64 to 128 registers provides little or no improvement in performance. [27:326] This example is also applicable to cache miss rates, regardless of cache associativity. [27:390-394] The point is that the benefits to be had by adding cache and registers is quickly exhausted.

Placing more types of functional units, or even including more functional units of the same type, has additional problems . First, the space available for FUs is finite. This means it is impossible to include all functions in a computer. Second, picking the functions that will result in the most speedup for all possible applications is difficult. Selecting new functions can be easy when the demand for those functions is high enough. An illustration of this is the recent introduction of the MMX Pentium processor. Third, functions that are not needed by an application go unused. In this case the space these functions consume is wasted since the area might be better used for other functions more suited to the application. Fourth, dealing mostly with duplicated FUs, the amount of

instruction level parallelism in a program is unpredictable and usually quite small. The benefits to be had from creating a very large VLIW machine (aside from the sheer complexity involved) is probably extremely low. Fifth, and last, there is only a finite number of instructions available in a computer's instruction set. Today's RISC processors typically have instructions that are only 32 bits long. Of those 32 bits, only a certain portion can be used to specify an instruction since many of the bits must be used for registers, addresses, or immediate values needed by the instruction. Adding more instructions would quickly consume the instruction set and necessitate changes to the computer's architecture.

RCs have the hope of overcoming the aforementioned five drawbacks to adding FUs. The reconfigurable nature of the FPGA means that the number of functions that can be implemented is almost infinite—there is a limit to the size of the function an FPGA can hold, of course. The FPGA can be configured to match the exact needs of every application. Every function in the reconfigurable hardware is needed, not wasted. The FPGA can match to the instruction level parallelism of an application. Instructions for the FPGA portion of the computer don't have to conform to the structure required by a computer's instruction set. The Chimaera project [24] illustrates this idea.

3.2.3 Summary of Needs

The analysis of the last two sections of the need and opportunities available for RCs is inconclusive. It appears that RCs can definitely be applied to increase the

performance of programs with certain characteristics. However, many programs don't have these characteristics and are less likely to experience significant benefits. Thus the range of applications that could benefit from reconfigurable computing appears limited. Still, continued research may expand the range of programs that can make use of reconfigurable hardware. Unfortunately, the exact magnitude of the benefits to be had and the number of applications that can benefit is difficult to estimate. It is also difficult to predict the exact gains obtained by adding registers, cache, or functional units.

Although this analysis is inconclusive, it appears likely that there are a significant number of applications which would benefit from reconfigurable computing, even if the reconfigurable hardware was included at the expense of alternate technology. The only way to further explore and genuinely understand reconfigurable computing is to build a reconfigurable computer. And, the first step in building any system is to define requirements. This is the subject of the next section.

3.3 Requirements and Design Considerations

Up to this point RCs have been discussed in the most general sense; no assumptions or analysis has been given about a design best suited to general-purpose computing. However, if a design for a general-purpose RC is to be proposed, the focus must be narrowed and some decisions about the design must be made. This can be accomplished in two steps. First of all, develop an initial framework for the design by ruling out designs based on the analyses of past RCs, FPGA strengths and weaknesses,

and the FPGA application base. It will also help to take into account subjects not already covered such as the practicality of introducing an RC system into the marketplace. All of this is done in this section. In the second step, more detail is added to the initial framework to provide a working description of a general-purpose RC. This will be covered in Chapter 4.

Perhaps the easiest way to develop the framework for the final RC is to list important design considerations and features required for a general-purpose RC. This list will enable high-level design decisions to be made. Once the framework is in place, more consideration can be given to the actual workings and detailed design of the RC.

- **A traditional processor must play host to the reconfigurable logic.** There are two very good reasons for this.

  First, a reconfigurable computer without a host processor must rely solely on reconfigurable components for all of the processing. The two options for a system consisting of a reconfigurable processor are a system that reconfigures for each application (a true custom computing platform) and a processor emulator, albeit one in which instructions can be changed or redefined at will. Both options are unfeasible. As has been demonstrated already, reconfiguring for each application takes a long time and requires a hefty investment in engineering and programming for each application. The overhead, both performance-wise and development-wise is therefore extreme. For a processor emulator, experience has

shown that emulators cannot deliver the performance of a fixed logic processor except for those tasks best suited for an FPGA.

Secondly, the transition from traditional computing to reconfigurable computing cannot be too extreme. This means that RCs must, for at least some period of time, continue to support the applications designed for traditional processors. Furthermore, since FPGAs are suited for certain types of applications, a computer relying solely on FPGAs for its processing would be ill equipped to handle the breadth of applications that might be presented to it. Conversely, modern processors are designed to handle just about any application. This, in turn, implies the von Neumann architecture of computers must be honored while adaptations are made for reconfigurable components. RCs are likely to fail commercially if pre-existing programs won't run on a reconfigurable computer, or even if users experience poorer application performance for existing applications. With this understanding, the host processor a necessity in today's environment. Moreover, the host should be of the most modern type available. This implies either a super-scalar or VLIW architecture. This is in contrast to many past efforts in which the host was either a CISC or single-issue RISC machine.

- **The FPGA needs a high bandwidth, low latency connection to memory.** The application base for FPGAs strongly suggests that FPGAs are most suited for regular, parallel tasks where the data is fed in a steady stream to the FPGA.

Although there is some leeway in implementation, it is always desirous to provide the best connection to memory possible. This is further supported by observing past RC designs in which the connections to memory were bottlenecks in performance.

The concept of "memory" is still vague at this point. The FPGA can be connected to memory in three ways: to main memory, to cache, or to the host's registers. If the RC is to follow a PFU approach, it makes sense to connect the reconfigurable logic to the host's registers. A coprocessor design, in which the FPGA may execute independently of the host, makes connecting to RAM or cache more appealing. An argument can also be made for connecting the FPGA to both registers and RAM or cache since it may be advantageous to use the FPGA in both the PFU and coprocessor roles.

- **The sizing of the reconfigurable hardware is driven by performance but constrained by cost.** Adding reconfigurable components to a host processor definitely impacts its cost. This is true regardless of whether the FPGA is included on-die with the host or on a separate chip, or chips. Bearing this in mind, the size of the reconfigurable hardware, and any support circuitry, should be kept to a minimum. Unfortunately, while less reconfigurable hardware is cheaper, it may mean sacrificing performance since a larger amount of logic can hold more, and bigger, functions. A design must balance anticipated performance gains with the unavoidable increase in price.

- **The host and reconfigurable hardware must interface so as to reduce communication overhead and protect program correctness.** Keeping communication overhead to a minimum maximizes the performance of the reconfigurable computer. As already explained, there is already a substantial overhead for using the FPGA hardware. Any addition to a superscalar processor must ensure that program correctness—instructions are completed in order—is maintained. This implies that data dependencies are detected and dealt with, that exceptions are handled correctly, and that data forwarding is addressed.

  A RC of the PFU type would naturally fit in well with a dynamically scheduled processor. If FPGA instructions could be designed to have source and destination registers like other instructions, then data dependencies could be handled as with any other instruction and FU. The processor would be able to perform register renaming, operand forwarding, and track FPGA instructions in its reorder buffer.

  If the chosen host processor is a VLIW machine, the PFU model would also be an attractive choice. The only drawback to a VLIW machine is that the performance of the processor depends greatly on the compiler's ability to schedule instructions and resolve data dependencies during compilation. In a normal VLIW processor (one without reconfigurable components) the compiler can produce highly optimized code because the compiler has full knowledge of the machine—instruction latencies, availability of functional units, and so forth—and

visibility over the code being compiled. In an FPGA-augmented VLIW machine, instruction latencies and the state of functional units may not be known during compilation. To compile for a reconfigurable VLIW computer the characteristics of the FPGA functions must be known in advance. Furthermore, the characteristics cannot change once the program is compiled or else program correctness may be jeopardized. This is not so for the dynamically scheduled machine since the hardware can reschedule instructions at runtime. For a dynamic machine the possibility exists then of being able to change an FPGA function without recompiling an entire program. A VLIW machine is still a viable platform, it is, however, slightly more inflexible than a dynamically scheduled processor.

- **The FPGA must be partially reconfigurable and, preferably, dynamically reconfigurable.** Little explanation is needed here since the advantageous properties of partial and dynamic reconfiguration have already been explained. Partial reconfiguration allows the FPGA to be shared by several functions at once. Perhaps not all functions are active at one time, but at least the potential exists for more than one to share the FPGA simultaneously. This leads to the idea of caching unused FPGA instructions, so that they are quickly available when needed again. Partial reconfiguration also allows the FPGA to change its functions during a program's lifetime so as to support the program during all phases of its execution. Dynamic reconfiguration simply adds to partial reconfiguration by

allowing functions to remain in execution while inactive portions of the FPGA loaded with new netlists.

- **Special enhancements to the FPGA design might have to be made.** Until recently, FPGAs were designed so as to be used in a wide range of tasks as possible. Building them this way made sense because users needed to be able to adapt FPGAs to widely varying tasks which the designers could hardly anticipate in advance. In RCs, the intended uses for the FPGA are more narrow and the opportunity exists to modify the basic design accordingly. The Chimaera and Garp projects are examples of instances in which the basic FPGA design has been modified to meet the needs of the computer being constructed. Several architectural changes seem readily apparent.

  First, it may be possible to decrease the extensive routing found in typical FPGAs. A system, like a computer, where the movement of data and control is usually in one direction may not need as much signal routing. Add to this that compute operations are typically done in a bit-wise manner and the demand, especially for long distance lines, tends to decline.

  Secondly, more capable logic functions may be included in each logic block or as special FPGA-only accessible FUs. Typically, FPGAs don't include support for addition or bit-wise comparisons and instead synthesize these operations from the simple logic in their logic blocks. However, special hardware support for these sorts of operations may be desirable in a RC. Instead of adding

functions to individual logic blocks, another idea is to place highly capable FUs throughout the FPGA. Functions running in the FPGA could then access these units to perform special operations on their behalf. The latter idea is rather radical—it probably puts too many constraints on the design of the FPGA and any functions that could be implemented in it—but it serves as an illustration of the freedom with which the can be FPGA designed.

A third change finds roots in the idea that the FPGA array must communicate with the host processor and fit into the host's execution model. It makes sense to provide the FPGA with a standard way of interfacing and interacting with the host processor. For example, the FPGA may need to signal the host that an error occurred. While the detecting of execution errors would be the responsibility of the FPGA function itself, the actual error signaling could be done by a fixed piece of hardware that implements some sort of error handling protocol. The concept here is that while the logic has to be reprogrammable, other parts of the FPGA array don't have to be.

One can conceive of several hard-wired services that the FPGA might provide on behalf of the functions loaded inside it. In addition to error flagging, other possible services include such things as informing the host that a function has completed execution, detecting that a function is ready for new data and then routing data to the function, and detecting and managing functions' requests to access memory. These services, in effect, constitute a standard interface between

the FPGA functions and the host processor. Providing a standard suite of services that functions can tap into is beneficial in at least two respects. First, the function designer can concentrate on the function's logic and not on integrating functions into the host's operation. And second, since functions don't need to program these operations themselves the size and, accordingly, the time load a function are reduced.

- **Special hardware might have to be added to the host system to support the FPGA.** In many respects this is similar to the concepts discussed in the previous bullet. Here, however, the extra circuitry operates independently of the functions loaded in the FPGA and instead helps the host system to maintain control over the FPGA. An obvious example of this is the loading of FPGA netlists. One can conceive of specialized hardware that could configure a section of the FPGA automatically with a specified netlist. The hardware would handle the reading of the netlist from memory and placement of the netlist into the FPGA's configuration memory. The presence of such circuitry would free the processor from this time consuming task, thus allowing the host to perform more important work, a definite advantage in a multi-tasking environment. Continuing in this vein, other additions might include such things as hardware acceleration of function placement (deciding if there is room to add a function of certain dimensions) and tracking the location and status of loaded functions.

Besides adding brand new features, it may also be necessary to augment features already present in the host. This stems from fact the FPGA could potentially expand the amount of work that a processor can perform. Increasing the number of registers or adding a special set of FPGA-only registers is an illustration of something that might prove beneficial. If the host is dynamically scheduled, then increasing the size of the reorder buffer and issue windows may be desirable. It may also pay to expand the number of result busses (sometimes called the common data bus) since the pressure to move results from the units producing them could rise with the FPGA in place. The FPGA could also increase the demand for reading and writing to memory. To meet the FPGA's I/O demands it may be necessary increase the number of reads and writes performed in one cycle. If the FPGA is allowed to go to memory directly, the demand is lessened. However, the processor design must still provide a method and resources that allow the FPGA to access memory on its own.

- **Instructions for FPGA functions must fit within the host's instruction set.** Invariably, the presence of new operational units, as the FPGA would become, necessitates the creation of new instructions for the host processor's instruction set.

Most RISC processors have a fixed instruction size—usually thirty-two or sixty-four bits—for all instructions. Furthermore, there are limited formats that instructions can take. For instance, a processor might have one format for

register-to-register instructions and a slightly different format for immediate instructions. Also, the total number of instructions that a processor can have is limited. Fortunately, most processors leave a few instructions available for future use. It is from this pool of unused instructions that the instructions for the FPGA will come.

The FPGA instructions are constrained by the three limitations described in the previous paragraph: instruction size, instruction format, and number of free instructions. Keeping the new instructions within the size limit ensures that minimal changes will have to be made to the host's instruction decode logic. That is, the decoder will have to recognize FPGA instructions, but won't have to deal with handling instructions of various lengths. Trying to place FPGA instructions within the format of instructions presently supported by the machine also reduces the amount of work needed on the decoder. If it seems necessary to create a new instruction format for the FPGA, the new format should be as similar as possible to existing formats for much the same reason. Finally, the number of new instructions should be kept as low as possible. Doing so leaves instructions available for future changes to the processor. It also means that controlling the FPGA will be done via a few, powerful commands, thus keeping the programmer's model of the FPGA simple to understand and apply by both the programmer and compiler.

- **The RC design must result in a predictable, easy-to-understand programmer's model.** This issue is mostly one of aesthetics, but has practical considerations as well. A design which is simple and graceful, while remaining powerful, has a greater chance of being accepted than is one which is awkward to use and difficult to understand even though it has tremendous performance. From a more practical aspect, a good design reduces the effort needed to learn a new system and use it. This, in turn, as alluded to in the previous section, can reduce the work of the compiler and programmer. Most important is creating a design that can be effectively programmed and scheduled by a compiler (or some other piece of software), thus relieving the programmer of the burden of ensuring proper program execution.

- **The RC design must consider OS, multi-tasking, and compiler issues.** It would be relatively easy to combine reconfigurable hardware on the same die with a conventional computer and get both to interact. Unfortunately, to make such a system practicable in a modern computing environment requires extra thought. The RC must aid or support more than just its own operation. An example of what is meant here can be taken from existing processors: paged and protected memory, context switching, a special class of instructions useable only by the OS, test-and-set instructions for protecting shared variables, and cache coherence protocols.

In a similar manner, a RC must furnish resources to support the reconfigurable hardware. Some of this has been touched on before, especially the creation of a simple design that can be used effectively by compilers. A chief compiler concern might include a way to guarantee program execution when FPGA resources are not available. Is there a way and what is the method for running a program in the absence of the FPGA? How does this impact the compiler? For the OS, issues to consider are context switches, loading and removing functions into and from the array, tracking process ownership of functions, fair sharing of FPGA resources among processes, and preventing malicious tampering with the array.

The overall point here is that there are aspects to success beyond simply combining an FPGA and conventional processor so that they form an RC.

IV. A RC Design: FPGADLX

4.1 Introduction

Chapter 3 explored the rationale and presented some high level requirements for the development of a general-purpose RC. This chapter builds on the concepts and notions developing a solid, workable RC design. This design is general enough in nature to be added to any present superscalar processor (and with some limitations to a VLIW) so as to turn it into a RC. Moreover, the requisite changes interfere minimally with the existing processor design and operation except for the unavoidable introduction of the FPGA array itself and its support circuitry.

The design unfolds in four stages. The first stage selects a high-level design from several models. Stage two is a description of a generic superscalar processor that will be modified to become a RC in the third stage. The last stage consists of a number of enhancements to the basic design. These improvements would result in a more aggressive RC, if implemented.

The RC architecture and operation described in this chapter were not created in a vacuum. There is, of course, more to any modern computer design than the architecture alone. In particular the compiler and OS are just as much a part of the overall RC as the architecture. Unfortunately, it becomes very confusing to discuss both aspects of the

design at the same time. For clarity, Chapter 4 concentrates on the RC's physical design and operation, while Chapter 5 covers the compiler and OS.

4.2 High-level Design Selection

Chapter 3 reviewed the positive and negative results of past reconfigurable computing efforts and defined a set of loose requirements for a general-purpose RC. These requirements now guide the design developed in this chapter. The design and evaluation process begins at a high, abstract level because it was felt that the best way to approach the design was with a macro-architecture that would act as a framework for the rest of the system. Determining early the placement and interaction of components such as the FPGA, CPU host, cache, and RAM so as to maximize the performance of the overall RC design, should result in a better system than if the FPGA was designed first and the rest of the system built around it. Indeed, as has been shown, too much of the RC's performance depends on the organization and coordination of the entire system, not just the FPGA.

Before describing the FPGADLX design a few design ground-rules derived from previous chapters are summarized.

1. For a general-purpose system, anything architecturally resembling the custom computing machines or the loosely coupled co-processors are undesirable. In a nutshell, this means that the host processor and reconfigurable logic cannot

connect over the system bus. Accordingly, some version of the closely-coupled co-processor or programmable functional unit classes of RC provide the best hope of succeeding. Justification for this decision was given in section 2.4.1.2.3.

2. The FPGA size should be equal to or less than that of a commercially available FPGA chip. A small size keeps cost down while still providing a decent amount of reprogrammable logic.

3. The RC system will include a host processor. The FPGA and CPU host should be on the same die or, at least, in the same package. The high speed of modern CPUs requires a close coupling of all components for maximum performance. The RC should not be a exception. This further motivates item 1.

4. The FPGA will need some amount of support circuitry (to load netlists, for example) in order to operate. This circuitry is considered part of the FPGA array and is not always mentioned explicitly in the discussion to follow.

Using these six rules as a starting point, the design process continues by looking at block diagrams of a set of four reconfigurable computing systems. The models are presented one by one with a short description given for each. Following that, the models are evaluated in terms of anticipated performance and by how well they conform to the rough requirements presented in Chapter 3. The models are listed in no particular order and the six rules listed above apply to each. All four models are alike in that they consist of a conventional computer system—CPU, cache hierarchy, and system bus—with

reconfigurable components and perhaps some support hardware added to it. This is about the only commonality among the four, however.

4.2.1 Model One

The first model is shown in Figure 11. The left side of the figure resembles a conventional computer system; the CPU is connected to some amount of cache which is, in turn, connected to the system bus and main memory. To this system an FPGA has been added. The FPGA can access the system bus and may optionally have its own cache.

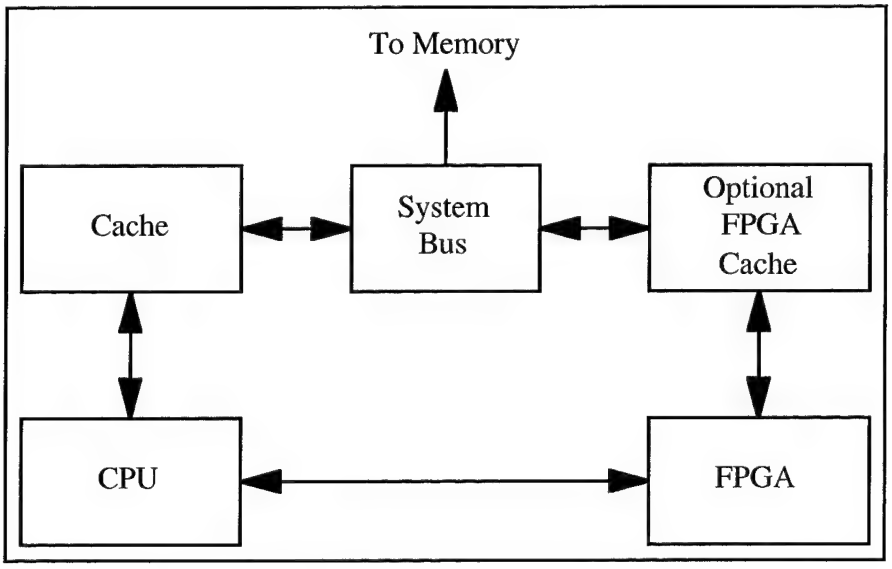

Figure 11. Model One. The FPGA can access the system bus independently and has its own cache (optional).

4.2.2 Model Two

Comparing Figure 11 to Figure 12 shows how, in this model, the FPGA's cache has been taken away. However, the FPGA shares the CPU's cache structure (at some level) and has been outfitted with its own local store. The local store is intended as a scratch pad for use by the FPGA when executing complex functions. The FPGA can access normal memory via the cache in order to load and store data as needed.

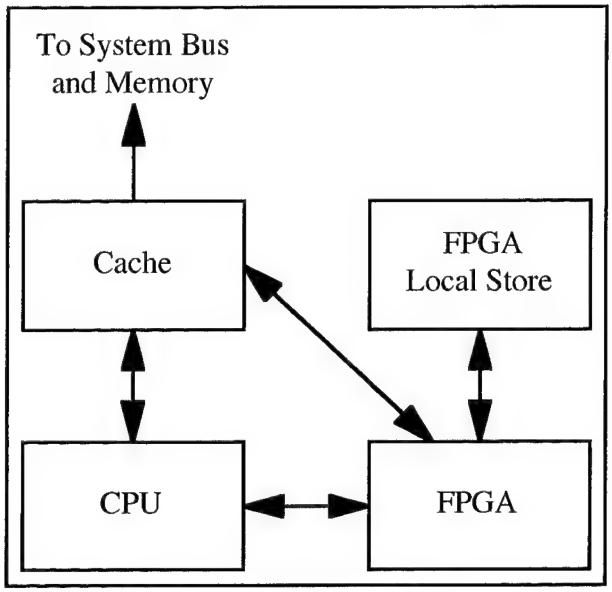

Figure 12. Model Two. The FPGA and CPU share the same cache while the FPGA is outfitted with its own local cache/memory.

4.2.3 Model Three

This is the same as Model Two, except that there is no local store for the FPGA. Figure 13 shows this kind of system. The absence of the local store limits the complexity of the functions the FPGA can perform since there is no place to keep temporary results. This does not preclude the FPGA from using cache or RAM to store data. However, because the CPU and FPGA share the same cache, the pressure on the cache, both in terms of access to it and in data collisions, will increase.

4.2.4 Model Four

Model Four is Model Three with the link between cache and the FPGA removed and is shown in Figure 14. In this model, the FPGA has pretty much devolved to its lowest level of connection; the only way for the FPGA to get data is through the CPU.

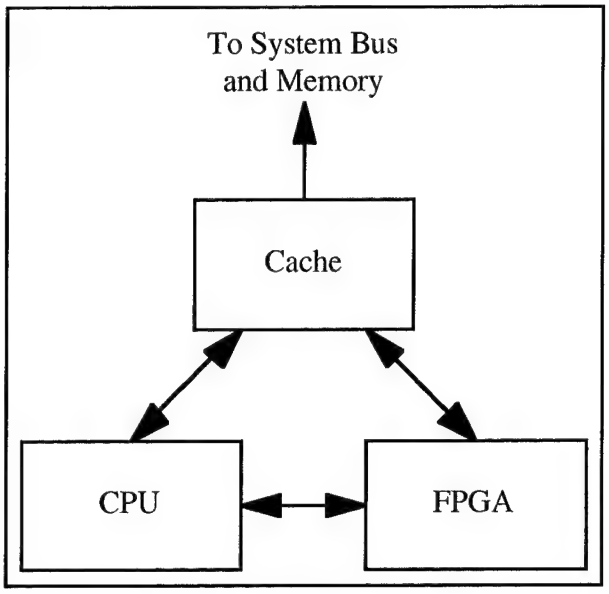

Figure 13. Model Three. The FPGA shares the memory with the CPU and has access (at some level) to the cache hierarchy.

4.2.5 Model Comparison and Evaluation

Now that the four models have been described, their features can be discussed and critiqued. The goal here is to eliminate unfavorable models and select a design which appears to hold the most promise.

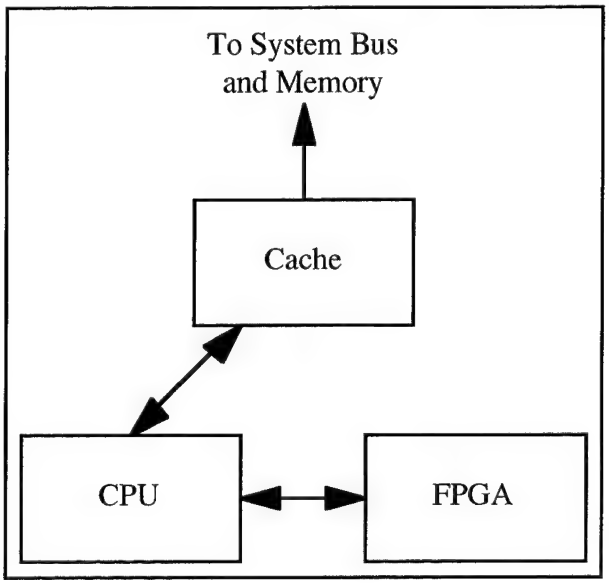

Figure 14. Model Four. The FPGA can no longer access memory on its own. Instead, it depends on the CPU for all of its data needs.

The first three models allow the FPGA access to the memory structure independent of the host processor. In the case of Model One the FPGA and the CPU host both maintain their own caches which are linked to a shared main memory. While the double cache alleviates the pressure on the cache, as

might be experienced in models Two and Three, there is one major drawback: the dual cache requires a cache coherence scheme to ensure data correctness. For general-purpose reconfigurable computing, implementing cache coherence mechanisms is probably unwarranted. Dual caches may increase performance, however, simply increasing cache size or associativity might also bring benefits without the troubles of enforcing cache coherence. The presence of dual caches weighs heavily against Model One.

Models Two and Three are the same except that Model Three lacks the local store. The local store was included as a design feature in response to the experiences of past researchers who found it helpful to provide the FPGA with some amount of exclusive memory. Characteristically, the projects in which this was done had FPGAs running large, complex functions. With the limited size of the FPGA in this developing RC design a local store is unlikely to be needed since the size of the expected functions will tend to be small. Certainly, they will be less than one FPGA chip in size! Any information storage needs can most likely be met by using some of the FPGA array's logic blocks as memory. While it cannot be said that a local store is completely unnecessary, it is not absolutely critical either. For this reason Model Two is removed from consideration.

The only difference between the two remaining models, Model Three and Four, is the ability of the FPGA to access memory directly. No matter which model is chosen, the final design must consider having increased I/O capabilities over that required by a

processor in a non-reconfigurable system. This is simply due to the fact that it is already difficult to keep modern processors supplied with enough instructions and data to keep them running at full capacity. Adding an FPGA to the processor makes the job even tougher. Methods of keeping a RC supplied with adequate amounts of data will be covered in depth in a later section. What needs to be debated presently are the merits of allowing the FPGA memory versus having it go through the host processor for all of its data.

The main advantage in letting the FPGA access memory directly is that it can do work more independently than if it had to rely on the processor for its data needs. Such capability would allow the FPGA to serve as a second processor for some tasks. The potential then exists for the FPGA and host processor to work, in some fashion, as parallel processors. As with many multi-processor systems, however, attention must be paid to maintaining program correctness wherever shared variables are concerned. This places the programmer and compiler in the position of having to protect critical portions of code—code that could be executed at the same time by more than one processor—against improperly ordered data reads and writes. In parallel processing this is known as code *synchronization*. At the present time synchronization requires that library functions or compiler directives be included in the source code so that shared variables in the code are handled properly. To some extent, compilers can automatically generate parallel code. However, they have limited intelligence and often miss many opportunities to produce the best code possible. Another factor to consider is that current parallel environments

consist of multiple, logically, if not physically, separate processors, each with their own cache and ports to the system bus. Conversely, Model Three would have two processors sharing these resources. It is unlikely that existing parallel programming techniques will be immediately applicable.

Another strike against direct memory access is that a good deal of extra hardware might be required. This is because the FPGA must have a method of generating memory addresses, initiating the memory access, and then receiving any results. How to implement such a scheme is unclear, especially when the FPGA plays host to multiple functions each of which may want to access memory. Providing functions access to memory when the number and location of functions is unknown until run-time presents a large problem. Further, it seems a waste of logic blocks to have each function be responsible for its own address generation.

From this discussion it seems that unsupervised, free-wheeling access to memory by the FPGA, as represented by Model Three, is less than ideal. However, is the alternative, relying on the host processor to handle the movement of data, any better? In at least two respects, yes. First, by forcing all data to move through the host processor the problem with shared data is eliminated. Because all data must be stored or loaded by the host, a single thread of execution is ensured. Unfortunately, this creates more of a bottleneck on the host's memory system. However, this can be partially alleviated by increasing the number of reads and writes permitted per clock cycle. The second benefit

is that the host already has the hardware needed for memory transfers. That is, the host can calculate addresses, initiate loads and stores, and handle any results or errors. This means that less hardware will have to be added to support the FPGA's data needs, than was the case with Model Three.

The best model appears to be Model Four. Other models may have some possible advantages over Model Four. However, the trouble or expense associated with them was deemed too high. Model Four is not perfect either; there is still the memory bottleneck to overcome. For now, though, it is assumed that this problem is lessened somewhat by allowing more memory accesses per cycle.

4.3 A Generic Superscalar Processor Host

Earlier it was decided that a host processor was needed, but the type of processor and its capabilities were not described. The next few paragraphs explain why a superscalar processor was selected. After that, details about the host processor's design and method of operation is given.

4.3.1 VLIW vs. Superscalar

In section 3.3 on page 58 it mentioned that if RCs are to be accepted in the marketplace, they must compete against existing computers in terms of price and performance. They must also run legacy applications without sacrificing performance. This means that the host processor an RC should reflect the latest technology in

microprocessors—the multiple-instruction superscalar or VLIW designs. Basing an RC on older, simpler single-issue CISC or RISC processors is unrealistic since those processors have already been made obsolete by superscalar and VLIW techniques. Unfortunately, in this thesis is developing only one design which means that only one processor type can serve as host for the RC. Therefore, the VLIW and superscalar qualities pertinent to reconfigurable computing must be compared and a winner selected.

Both VLIW and superscalar techniques are suitable from the standpoint that they represent the types of processors found in contemporary computer systems. In this respect, either type will serve as a good base for adding reconfigurable components. However, as discussed in section 3.3, the VLIW scheme has one quality that makes it slightly less desirable than a superscalar design. In short, the drawback is that to compile programs for a VLIW computer, the exact properties of the machine must be known at compile time. While adding reconfigurable components does not prevent the nature of the RC from being known, it does hinder the freedom with which the RC can be used. For instance, say a reconfigurable program is compiled and that it uses the FPGA to compute a result that will stored a register. The FPGA function takes eight clock cycles to compute. The program can be compiled because the latency of the FPGA function is known. However, the program couldn't be compiled at all without knowing the latency because the compiler wouldn't know how to properly schedule instructions. Moreover, assuming that the program was compiled, the program would have to be recompiled if the latency of the FPGA function were to change. This problem does not apply to

superscalar machines since register renaming and operand forwarding make the latency of operations a non-issue.

All things considered, the drawback to using a VLIW is relatively minor and either design is suitable. However, the decision to pick a superscalar processor was based on factors unrelated to the merits of either design. These factors center on being able to test and evaluate the eventual RC design. The first is the development of a RC simulator. The choices were to develop one from scratch or, preferably, to modify an existing simulation. Fortunately, there were existing programs available which modeled various superscalar machines. No VLIW simulations could be found. The second factor was finding a compiler that could be reworked to produce instructions which could be used with the selected simulator.

One complete package, a superscalar DLX [27, 41] simulator bundled with a DLX compiler, fulfilled both requirements. The DLX processor is known as SuperDLX. The only shortcoming of picking a DLX processor is that it is not a real processor; the DLX instruction set is primarily used as a teaching tool. However, considering the tight budget constraints of this project (i.e., no money), the DLX simulator and compiler were attractive because they were free. Furthermore, the source code is freely available which means that both can be modified at will. The compiler and simulation are discussed in great detail in Chapter 5. The DLX processor design is covered in the next section.

4.3.2 The Architecture of the Superscalar Processor Host

As mentioned in the previous paragraph, there is no actual superscalar DLX processor. This presents a problem since the ideal situation would be to take a real design, turn it into a RC, and then see how well the RC performs compared to the existing design. Unfortunately, that opportunity did not materialize. However, the situation is not as dismal as it may seem.

First of all, the DLX instruction set forms a major subset of several prevalent superscalar processors including the PA-RISC, PowerPC, MIPS, and SPARC. [27: Appendix C]. Secondly, because a simulator is being used, the freedom exists to construct a processor model that mimics one of the aforementioned processors or even merges features of several of the processors. Additionally, studies can be easily performed on the effects of changing the design and operational characteristics of the processor, e.g. varying the number, types, and latencies of functional units. For this project it was decided to include features found in actual processors, especially if those features were already in the DLX simulator, and to fix those features for testing purposes. The idea was that the resulting model would be generic enough so that the eventual RC design could be adopted by any superscalar processor and that the test results would be representative of what one might expect if an actual processor design had been used. This fixed-function superscalar host is described in the remainder of this section.

SuperDLX contains architectural features and performance characteristics derived from several late-generation superscalar processors. The processors examined include the HP PA-8000 [31], PowerPC 601 [46], MIPS R10000 [39, 40], and UltraSPARC [49]. Specific attributes taken from these processors will be mentioned as the design is presented.

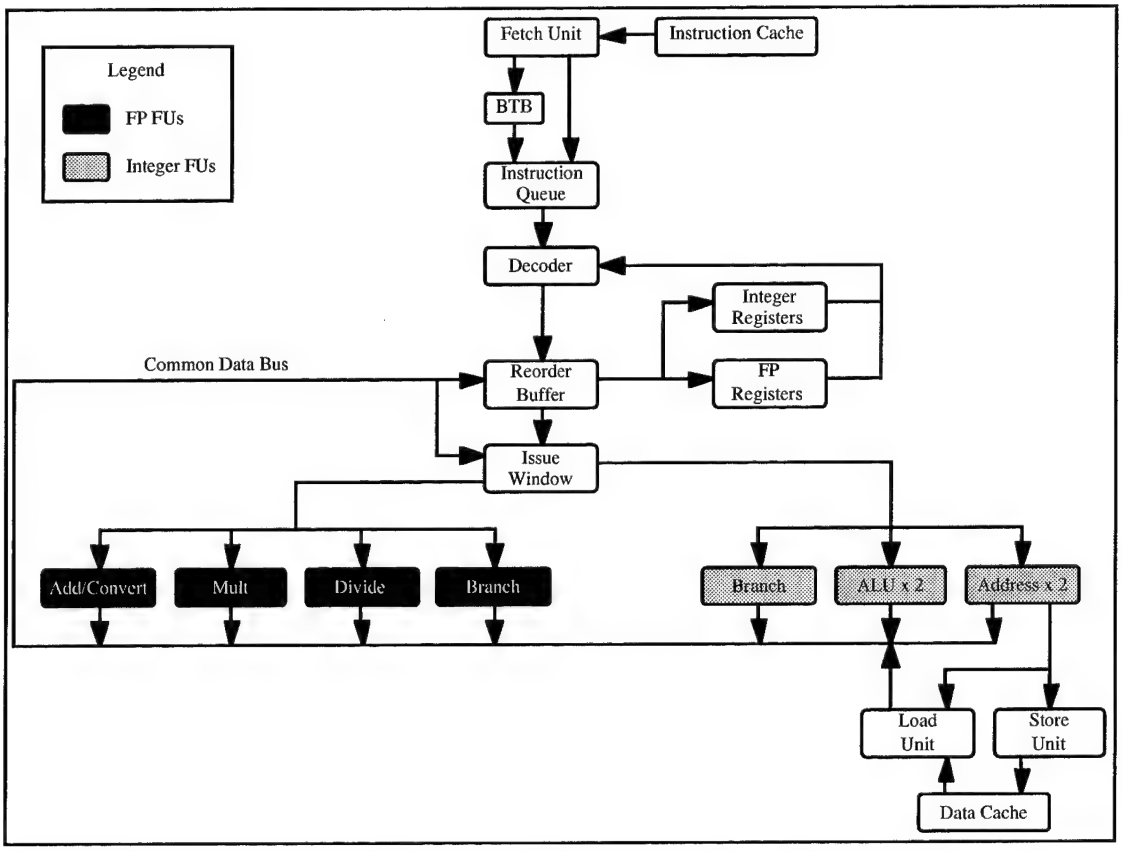

Figure 15. SuperDLX Block Diagram. SuperDLX is a superscalar implementation of the DLX instruction set and reflects many architectural features found in modern processors. The processor is four-issue machine and can support up to two memory accesses per clock cycle (not including instructions).

A schematic of SuperDLX is shown in Figure 15. The design is truly superscalar; it performs in-order instruction issue and completion with out-of-order execution. Out-of-order execution can occur because the processor dynamically renames registers so that

data can be forwarded to pending instructions as soon as it is available. The processor can issue and complete up to four instructions and perform at most two memory accesses per cycle. The processor executes instructions by stepping them through five or six stages depending on instruction type—loads and stores take six, all others, five. The six stages are fetch, decode, execute (sometimes called issue), write back, commit, and memory.

Major architectural components include a sixteen-entry instruction fetch queue, a 419-entry branch target buffer (BTB), a 32-position instruction issue window, a 32-entry reorder buffer, separate five-deep store and load buffers, and a suite of integer and floating-point functional units. The reorder buffer is implemented in hardware as a circular queue as shown in Figure 16. Each entry in the reorder buffer consists of five fields: type, address, destination, result, valid bit, flush bit, error bit, address bit. The type field identifies one of four destinations for the result of an instruction: integer register, FP register, memory, and none. The address field stores the address of the instruction. Destination supplies the register number where the instruction should be written. (Addresses for loads and stores are kept in the load and store buffers). The result field is used to hold the result of the instruction until the instruction commits. The valid bit marks that the instruction has written back and is ready for commit. The flush bit indicates that the instruction has been flushed and should not commit even though the valid bit is set. The error bit is set during write back to indicate that the instruction raised an exception during execution. The address field stores the address of the instruction.

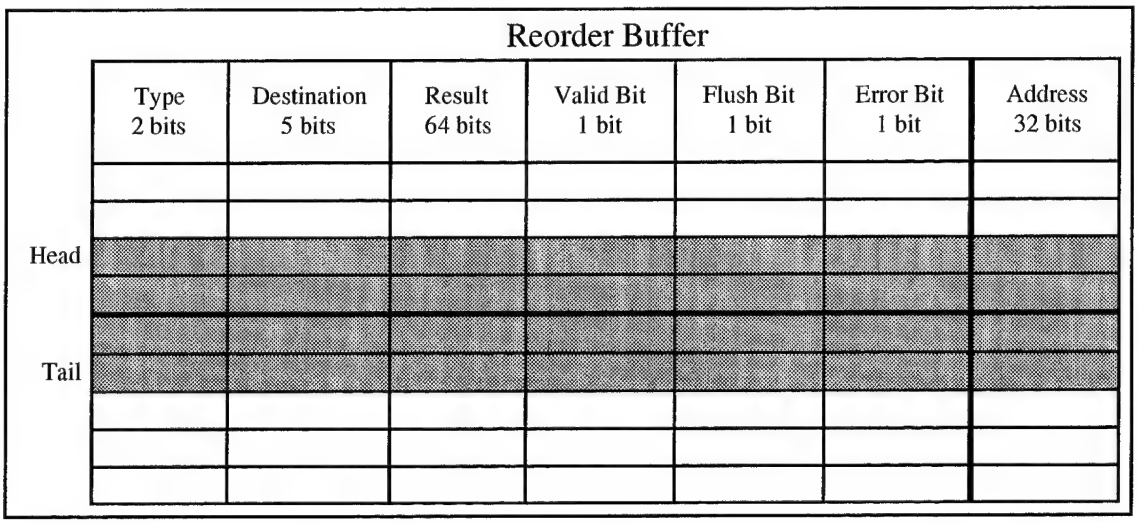

Figure 16. SuperDLX Reorder Buffer. The reorder buffer is a circular queue with 32 entries. (This diagram shows only a few of the entries). Each entry consists of five fields: type, destination, result, valid bit, flush bit, error bit, and address.

The nine functional units are:

- Two integer ALUs (add, subtract, shift, compare, and logical operations)
- Two address calculators
- One integer branch unit
- One FP adder/FP-integer converter
- One FP multiplier
- One FP divider
- One FP branch unit

All the FUs have a one cycle latency except for the FP adder, multiplier, and divider which have latencies of three, three, and 17, respectively. (Latency is the number

of clock cycles needed to compute a result). The integer latencies come from the MIPS R10,000 processor. The FP latencies are for single-precision numbers only and match those of the PA-8000 processor. Double-precision latencies are not included since the simulator, at present, does not distinguish 32- and 64-bit FP operations. This is not seen as an issue since the FPGA is not typically used in FP-heavy programs. The FP adder and multiplier are also pipelined and can accept a new instruction every clock cycle. The divider is not pipelined and can only perform one divide at a time.

The memory and cache hierarchy included in, or bundled with, a processor can have a major impact on performance. At this time, SuperDLX has no specific memory system. There are several reasons for this. First, the simulator assumes a perfect cache and lacks any cache simulation features. Adding a cache to the simulator would have been a time-consuming task. Secondly, designing a successful memory system is an art and is considered to be beyond the scope of this thesis. Third, the focus of this thesis is on producing a design for a general-purpose RC, not caches. While the cache is an important part of any processor, it is of secondary importance to the reconfigurable aspects of the design. Lastly, thinking ahead to the RC, it is not immediately clear if or how dramatically the cache factors into performance. As a result, cache issues are discussed only when it is believed that they impact performance enough to merit serious attention.

However, to aid in the presentation and development of the RC design, several assumptions have been made about cache and memory. The first is that there is some

amount of level one (L1) cache on die with the processor. L1 cache blocks are four words, 128 bits, long. This is in line with modern processors. The cache can load and store entire blocks at once. However, other cache characteristics such as associativity, size, and whether it is unified or separated are subject to debate. It is also assumed that there is an off-chip L2 cache of indeterminate capacity and relationship (look-aside, write-through, etc.) to the L1 cache. Unlike the caches, no assumptions are made about the RAM included in the system except to say that there is some. It is fairly safe to expect that a satisfactory memory system can be built to meet the needs of SuperDLX and its reconfigurable enhancements.

To comply with the DLX instruction set, SuperDLX has 32 integer registers, 32 floating point registers, a floating-point status register, and a register for the program counter (PC). Peculiar to DLX, all integer multiplication and division is handled by the FP multiplier and divider, respectively. SuperDLX operation is described in the next six sections.

4.3.2.1 Fetch Stage

In this, the first processor stage, up to four 32-bit DLX instructions are fetched from memory every clock cycle and placed in a 16-deep FIFO instruction queue. Most instructions are simply put into the queue, although a few instructions receive special handling:

1. NOPs are ignored.

2. TRAP instructions are used by the processor to make calls to the operating system. The fetch stage handles all TRAPs as normal instructions except for TRAP #0 which is used to signal the end of a program. Fetching TRAP #0 stops the fetch process permanently.

3. Branch and jump instruction handling depends on whether or not branch prediction is activated. The procedure for dealing with branches and jumps is detailed below.

   The machine has an optional branch prediction feature that makes use of a 419-entry target buffer (BTB). The BTB uses a four state prediction scheme. This prediction scheme is used in the UltraSPARC and R10000 processors. The default state for new branches is a weak TAKEN. Branch target addresses are also stored in the BTB. The BTB is updated as required as branches are executed. For taken branches the program counter (PC) is set to the target address as soon as the address is known and the processor begins fetching at the new PC. For untaken branches the PC is unaffected. Without branch prediction fetching halts any time a branch or jump is encountered. Fetching continues once the target address is known—after the address has been computed and the branch outcome is determined. Unconditional, direct jumps don't stop the fetch stage since all the information needed for them are contained within the instruction. With branch prediction turned on fetching occurs as described above except that fetching can

continue uninterrupted for conditional branches based on predictions stored on the BTB.

Supporting speculative execution means that the processor must be able to recover from incorrect branch predictions. The machine has the ability, therefore, to flush all instructions following an incorrectly predicted branch. Flushing consists of clearing the fetch queue, marking the reorder buffer entries of instructions on the mispredicted branch path as "flushed", resetting the PC to the correct branch address, then resuming the fetch process from the new PC address. Dealing with flushed instructions will be covered in the Commit section below.

4.3.2.2 Decode

Up to four instructions per cycle can be removed from the instruction queue and decoded. Decode occurs as long as there are valid entries in the instruction queue and there are open slots in the reorder buffer, the issue window, and, if the instruction is a memory operation, in the load or store buffer. Compared to the fetch stage, decode is rather simple. It consists of identifying an instruction's type, assigning the instruction an entry at the end of reorder buffer, and entering the decoded instruction into the issue window. Memory instructions are also entered into the load or store buffers. Instructions are decoded and entered into the issue window in program order.

Once in the issue window, the instruction will need copies of its operands so that it can eventually enter a functional unit for execution. (An operand is considered to be a source register number from which data for the instruction is supposed to come or an immediate value.) Immediate operands are simply placed into the issue window. Register-based operands require forwarding of the most recent operand value to the instruction. First the reorder buffer is searched. If the value is in the reorder buffer but has not been computed yet, the reorder buffer entry number is taken instead of the operand's value. The buffer entry number will allow the data to be forwarded during a later clock cycle. Data forwarding is done by snooping on the CDB as results are written back. If the operand has been computed and is in the reorder buffer, the operand's value is placed into the instruction's window entry. When the operand is not found in the reorder buffer, then its value is forwarded from the register. The instruction also needs its reorder buffer entry number. This number is stored in the issue window alongside the instruction's operands.

4.3.2.3 Execute

An instruction remains in the issue window until all of its operands are valid and there is an open functional unit of the right type to handle it. Instructions meeting these two requirements are removed from the issue window and begin executing in the appropriate FUs. The oldest instructions in the issue window are taken first. This prevents starvation. Up to four execution-eligible instructions are removed every clock cycle. Entering execution consists of taking the instruction's operands and reorder buffer

entry number from the issue window and placing them in the FU. The operands are used to compute the result. The reorder buffer number moves through the functional unit with the instruction. Once execution is finished, the number is used to identify which reorder buffer entry will receive the result during write back.

4.3.2.4 Write Back

In write back, results from the functional units are placed in their correct reorder buffer entry via the CDB. A maximum of four results are allowed to write back during one clock cycle. Results corresponding to the oldest reorder buffer entries have priority. The assumption is that old instructions have a greater chance of causing other instructions to stall waiting on their results. Once a result is written back the FU is free to produce another result.

4.3.2.5 Commit

The commit stage sends the results of the reorder buffer entries which have written back to their appropriate register. Upon committal, instructions are removed from the reorder buffer, thus freeing the reorder buffer entry for reuse. Instructions are committed in program order. Four instructions can commit on each cycle, but commit stops when an instruction that hasn't written back is encountered.

Recovering from mispredicted branches—handling flushed instructions—is also done in the commit stage. SuperDLX allows flushed instructions that have made it into

the reorder buffer—they made it through decode before the outcome of the branch was known—to execute. These flushed instructions will write back, also. However, because they are marked as "flushed", they will not be allowed to commit to registers. This manner flushed instructions are allowed to *drain* out of the processor by passing through all execution stages, receiving special handling only at commit time.

4.3.2.6 Loads and Stores

Loads and stores are treated differently than other instructions by the processor since their execution is accomplished in two phases: address calculation and memory access. The first phase occurs in either of the two address FUs. Once known, the addresses are forwarded to the operation's entry in the load or store buffer.

In phase two the loads and stores are allowed to go to memory. A load can begin memory access as soon as its address is known and there is no store older than it that accesses the same memory location. The latter condition is commonly referred to as "load by-passing" because it allows loads to move ahead of stores in execution order. This is a desirable feature since loads provide data that will soon be needed by other instructions, whereas stores are not so critical. Additionally, loads can to go to memory in any order from within the load buffer because their ordering has no affect on program correctness just as long as they do not by-pass older stores. The data retrieved for a load is placed in the load's reorder buffer entry in a manner similar to write back.

A second helpful feature called "load forwarding" is not currently supported by SuperDLX because it was not a part of the simulator. (It is not clear why this feature was not included by the simulator's original author. In any event, it would have taken too much time and been of limited benefit to add it for this effort.) In load forwarding, data from pending stores is forwarded to younger loads that have the same memory location. Forwarding allows the load to complete without actually going to memory or cache, thus conserving critical memory cycles.

Unlike loads, stores must proceed to memory in strict program order to preserve proper program execution. Therefore, stores are released to memory only when their corresponding reorder buffer entries are committed. Committal also places the data to be stored into the instruction's entry in the store buffer. Once released, stores move on to memory in FIFO order from the store buffer.

4.3.3 Performance Questions

SuperDLX has a more capable architecture than most actual processors. Real processors have to place restrictions on their hardware because providing every feature desired would cause them to be overwhelmingly complex and too costly to implement. The limitations burdening actual designs reduce the performance of those designs. Subsequently, SuperDLX is, potentially, a more powerful processor than any existing one. Having a stronger model than is currently achievable for a typical superscalar processor will tend to weaken the apparent performance of the RC to be built with

SuperDLX as its core. In a way this is good because, if the RC performs reasonably well on reconfigurable equivalents of test programs run on SuperDLX, then the reconfigurable hardware will be even more impressive when supplementing a weaker, real-life processor. The four items are briefly discussed here. The restrictions facing actual processors and how SuperDLX differs are discussed below.

- **Issue Window** — At least one processor design, the PPC, limits instructions to issuing from only a few positions in the issue window. Usually only the bottom few positions are eligible for issue even though other entries meet the issue criterion. Conversely, SuperDLX allows issue to occur from anywhere within the window.

- **Issue Logic** — Some processors such as the R10000 and PA-8000 have several issue windows each relating to a type of functional unit (e.g., integer, FP, and memory) and allow a limited number of issues per window every clock cycle. Even designs with a single issue window usually allow let a few instructions of each type issue. As an illustration, take the case of an integer program executing on a processor with a two integer, two FP issue limit. Since no FP instructions will ever be executed, the processor is prevented from ever issuing more than two instructions at a time. Only when FP programs are run can the number of instructions issuing per cycle climb above two. On the other hand, a mix of four instruction of any type are issuable under SuperDLX.

Real processor also place limits on the use of some of their functional units. For instance, a processor may have two ALUs both of which can do math and logical operations, but only one can do shifts. Whenever a shift occurs it can only execute in the one ALU. SuperDLX is also devoid of these types of operational limits.

CDB and Write Back Logic — The CDB, which moves results from the FUs to the reorder buffer, is typically restricted in terms of which of its channels serves which FU. As a result, write back of results may be delayed because a CDB channel is not available to a FU even though there may be free channels. Comparatively, all CDB channels are made available to every FU in SuperDLX.

4.4 The Basic RC Design

SuperDLX forms the base upon which the FPGA and its supporting hardware are added to make a RC. This section details the changes to SuperDLX to make it FPGADLX. Among them are such things as a new issue window, new assembly instructions, modified decode and write back logic, and last (but not least) the FPGA. The focus here is on the hardware and operational aspects of the FPGADLX. To illustrate the ideas presented, the design ends by reviewing the five stages of execution experienced by instructions using the new hardware. Support provided by the operating system and compiler will be discussed in Chapter 5.

Before delving immediately into things, a few words need to be said about what is meant by this section's title *The Basic RC Design.* One of the chief concerns in making SuperDLX into a RC, was that SuperDLX's design and operation be affected as little as possible and that new hardware be kept to a minimum. The idea was that by not tinkering with a proven computer design—in this case SuperDLX is a melting pot of architectures—too much, the potential for achieving a successful and elegant RC would be increased. Also, an overly aggressive initial RC design might overshadow the benefits of a simpler design making it impossible to determine the performance impact of a few small-scale changes. For the most part, the design conveyed in this section is responsive to these concerns in that it fulfills the requirements necessary to qualify as a reconfigurable superscalar processor, no more and no less. The design attempts to meet the requirements and addresses issues presented in Chapter 3 and in section 4.2. At every turn, meeting those goals while maintaining simplicity and performance was of paramount importance. Hence, the idea of a *basic* design.

4.4.1 Design Overview

FPGADLX is essentially a PFU-style RC built into a superscalar processor host. As can be seen by comparing Figure 15 to Figure 17, the reconfigurable hardware is grafted into the host at the most fundamental level. Integrating the FPGA and its support hardware into SuperDLX at the lowest level allows the new hardware to take advantage of the already powerful superscalar design. Indeed, FPGADLX has the same design and operates just like SuperDLX for all instructions in the normal DLX instruction set. The

main difference between SuperDLX and FPGADLX is that the latter has an expanded instruction set and new hardware.

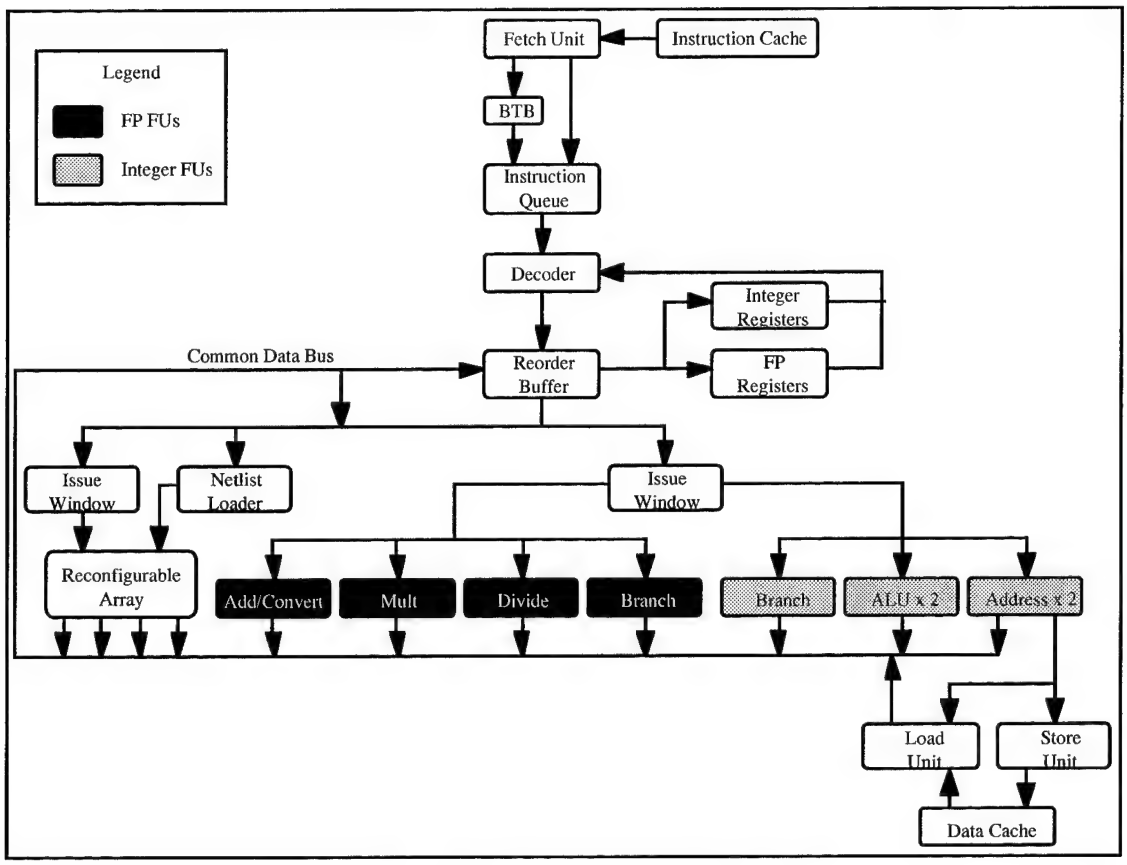

Figure 17. FPGADLX Block Diagram. FPGADLX is essentially SuperDLX with a new issue window, a netlist loader and reconfigurable array. The new additions become part of the processor's operation and make use of resources and services provided by the host processor.

Unfortunately, describing the additional hardware that transforms SuperDLX into FPGADLX can become rather confusing at times, especially when trying to explain how the new hardware fits into SuperDLX's execution model (fetch, decode, execute, write back, and commit) at the same time. Often times, a vicious circle ensues. To avoid potential confusion, a quick overview of FPGADLX architecture, instructions, and operation has been put together.

4.4.1.1 Architecture

- **FPGA** — The FPGA has been divided into four banks and is 64 rows deep. Each bank is 32 logic blocks wide and can accept two 32-bit input operands and produce one 32-bit result each clock cycle. The FPGA supports pipelined and non-pipelined synchronous functions as well as asynchronous functions and functions with variable latencies. The FPGA can support the simultaneous execution of multiple functions.

  Each row of each bank includes support hardware which assists the FPGA during execution and integrates the FPGA into the host processor. This support hardware is also reconfigurable in that its operation is determined by the needs of the function using it. The configuration for the support hardware is included as part of the netlist for a function.

- **FPGA Issue Window** — Supporting the FPGA, in which functions can accept up to eight input operands and produce four results, called for the construction of a special issue window. This FPGA issue window can store the data needed for up to five functions. Its operation is much like that of the normal instruction window from the perspective that issue can occur out of order and operands are forwarded in a like manner.

- **Operand Switch** — The operand switch consists of four 64-bit switches. The operands switch is used to align operands coming from the FPGA issue window with their correct banks in the FPGA.

- **Netlist Loader** — The fact that the FPGA needs to be configured prior to execution lead to the idea of the Netlist Loader. The Loader is a special device integrated into the processor and operated under the control of the OS. The Loader's job is to move netlists from the computer's memory, during otherwise idle memory bus cycles, into the FPGA's configuration memory.

4.4.1.2 Instruction Set Additions

Four new instructions were added to the DLX instruction set. The new instructions reuse existing instruction formats. Three of the instructions are normal in the sense that they execute in the hardware and pipeline found in SuperDLX. These new instructions cause netlists to load and unload, a process ID (PID) register to be set, and the Netlist Loader registers to be written. However, the fourth instruction, aptly named the FPGA execute instruction, causes functions loaded in the FPGA to execute. Since the FPGA is involved, the path and handling of FPGA execute instructions differs slightly from all other instructions.

FPGA execute instructions occur in sequences of up to four. A sequence specifies all the input and output registers needed to execute a function. Each instruction in a sequence corresponds to one bank of the function.

4.4.1.3 FPGA Functions

Traditionally, functions has been synonymous with a netlist. In FPGADLX, however, a netlist may contain several functions. It was realized that several functions may be needed to perform a task or implement the section of code that is to be executed in the FPGA. If netlists held only one function, then some functions risked not loading. Therefore, it is a netlist, and its functions, which are loaded into the FPGA's configuration memory.

Functions can be any number of rows deep and from one to four banks wide. A function's internal operation, including that of its support hardware, is totally determined by the netlist. Additionally, there are some rules governing the manner in which functions interact with the host processor which the netlist must honor. Functions are allowed to cooperate and share data in an arrangement called "cooperating functions." The cooperating functions concept was developed as a means to allow complicated functions to be built and work together on a task without having to introduce new instructions.

4.4.1.4 Superscalar Operation

Quite understandably, the introduction of new hardware and instructions to SuperDLX meant that the operation of the processor would be affected to some degree. In this case, the execution pipeline has been split into two parts—one part for the FPGA

execute instructions and the other for all other assembly instructions. A summary of the altered execution path is given below.

- **Fetch Stage** — The same fetch queue exists and still fetches four instructions per cycle. In every regard, fetching of instructions remains unaffected.
- **Decode Stage** — Decoding operates as normal with up to four instructions being decoded from the fetch queue and entered into the reorder buffer (the reorder buffer is reused) and issue window every cycle. However, FPGA instructions are fed into the FPGA issue window as a sequence so that one entry of the window corresponds to one invocation of a function.
- **Execute Stage** — When all of the operands needed for a function to execute are valid, a sequence may enter the FPGA and begin execution. Only one function may begin execution on a given clock cycle. On issue, the Operand Switch is configured so that the issue window lines up with the correct banks of the FPGA and the data in the window are made available for the FPGA to latch.
- **Write Back Stage** — During write back, the four banks of the FPGA are treated as if they were each a functional unit. Accordingly, when a function has produced results, write back logic arranges for each bank of the function to place its respective result on one of the CDB's four channels.

- **Commit Stage** — Extra logic is needed to commit FPGA execute instructions correctly during context switches. Other all other respects, commit is unaffected and FPGA execute instructions are handled in the same fashion as all others.

4.4.1.5 OS and Compiler

Putting together a successful computer design consists of more than working out the details of the hardware and instruction set. It also means giving consideration to making the system easy to use and manage. Thus, FPGADLX was designed with the compiler and OS in mind. However, FPGADLX also demands extra duties from the OS and compiler.

- **OS** — The OS is impacted in two ways. First of all, the OS is responsible for managing the state of the FPGA. This follows from the fact that the FPGA is a shared system resource and must be protected. The method worked out is for the OS to handle netlist loads and unloads for all processes. The second impact is that the FPGA has to perform a little extra work during a context switch.

- **Compiler** — Ignoring the many problems facing reconfigurable compilation in general, several specific jobs for an FPGADLX compiler have been identified: using and incorporating netlists into a program, determining when to load and unload netlists, and generating FPGA execute instructions so that functions are invoked properly.

With this mini-review of the FPGADLX design completed, a detailed explanation can commence.

4.4.2 FPGA Issue Window

The regular issue window's main purpose was to store and track the valid status of operands for individual instructions where, as far as the window knows, the instructions are unrelated. When all the operands for an instruction were valid the instruction was allowed to move into an FU for execution. The same remains true for the FPGA issue window except that the FPGA issue window stores and tracks operands for sequences of instructions relating to specific invocations of FPGA functions. When all of the operands in a sequence are valid, the function relating to the sequence is allowed to execute.

The FPGA issue window has five entries and operates as a collapsible stack. Thus, up to five pending functions can be stored in the window at once. An entry stores all the information needed to execute one function. As might be expected, the FPGA window entries are larger than their regular issue window counterparts. An example window entry is shown in Figure 18. The figure has been drawn to so that the operation of the window might be easily understood. Physically speaking, the entry would not be constructed as illustrated in the figure. Space for eight operands is required since up to four FPGA execute instructions are decoded for each function. As instructions are decoded, their operands, or the reorder buffer entry that will produce the operand, enter

the window in pairs. For example, the first instruction in a sequence has its operands placed in Operand 1 and Operand 2 while the second instruction's operands would go into Operand 3 and Operand 4, and so on, until the sequence completes. This concept is displayed in Figure 18 where entry columns are shown in relation to the order of instructions in a sequence.

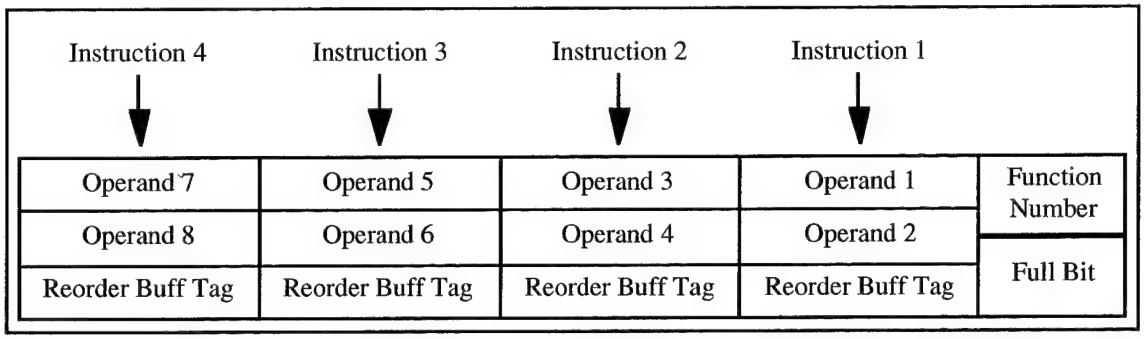

Figure 18. An Entry in the FPGA Issue Window. An FPGA issue window entry stores all the information needed to execute an FPGA function. This includes the operands and reorder buffer tags for up to four instructions, the function number to be executed, and an entry full bit. Not shown in the illustration are the valid bits associated with each of the operands. The illustration gives the concept of a window entry and is not meant to represent a physical layout or design.

Instructions always enter into their window entry in this right-to-left fashion. Decoding of new sequences always begins in one of the window's unused entries. Subsequent decodings within the same sequence continue in the same entry in which the first instruction in the sequence was decoded. A window entry is marked as full by setting its full bit when the end of sequence bit is set in one of the instructions. Only full entries are allowed to move into the FPGA for execution. For a full entry to move into the FPGA, all of its filled operands (those for which an instruction was decoded) must be valid. Once a full entry has entered the FPGA, its full bit is set to false, indicating that the entry is unused and is ready to accept another sequence.

Along with an instruction's operands, the entry also stores the instruction's reorder buffer tag. Reorder buffer tags must be saved since the instructions, even though they are in a sequence relating to one function, are still given their own entry in the processor's reorder buffer. The reorder buffer tags are passed to the FPGA when an entry begins execution. As with non-FPGA instructions, the tags will be needed in order to match results coming out of the FPGA (considered to be a functional unit) with the reorder buffer entries during write back.

The last field in the entry holds the FPGA function number. The function number is used for two different purposes. First, the function number is used to detect errors in FPGA instruction sequences. An entry's function number is set when the first instruction in a sequence is decoded and enters the window. The value stored is the ten-bit function number from the decoded instruction. The function number from subsequent instructions are compared to the number stored in the entry. A mismatch means that instructions for two different FPGA functions are attempting to use the same entry. This constitutes a fatal error, forcing the processor to stop execution on the process in which the error occurred. The second use of the function number field comes when the entry finally enters into execution in the FPGA. The function number stored in the field is used to find the function to be executed in the FPGA and to align the entry with the function.

4.4.3 The Reconfigurable Array

As mentioned several times before, adding an FPGA to the superscalar processor requires some amount of support hardware. Some of this support hardware is external to the FPGA, while the rest can be considered part of the FPGA itself. It's convenient to think of the FPGA and its internal support hardware as one unit—the reconfigurable array (RA, for short). The RA consists of two parts: the FPGA and an Operand Switch. The shaded areas in Figure 19 show how the RA relates to the FPGA issue window and the processor's common data bus (CDB). The FPGA and Operand Switch are described in the next two subsections.

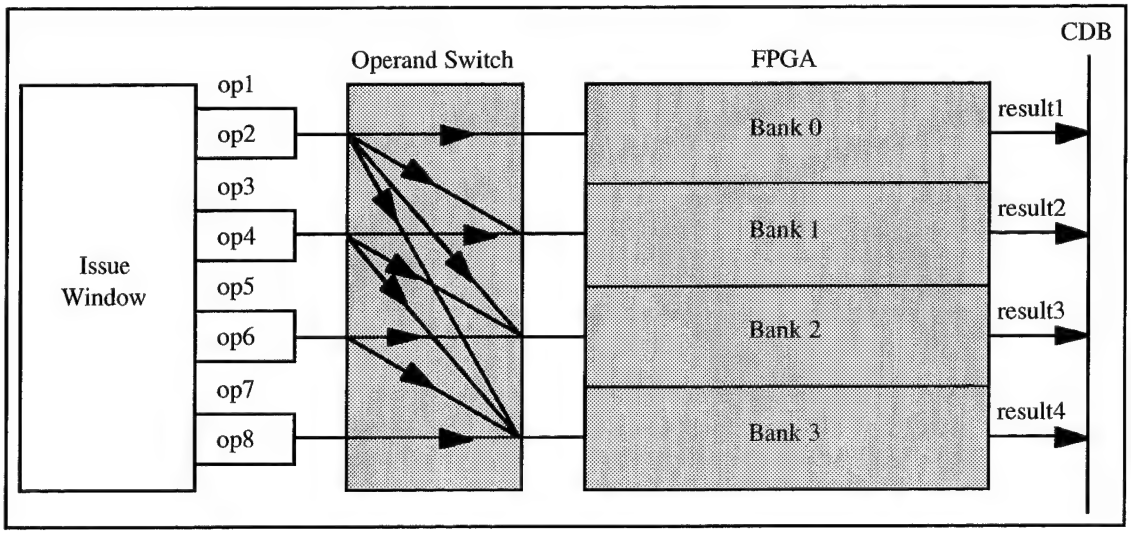

Figure 19. The Reconfigurable Array (RA). The RA consists of two main parts: the operand switch and the FPGA. The operand switch is responsible for aligning operands coming from the issue window with the correct bank(s) of the FPGA. The FPGA consists of four 32-bit wide banks. Functions loaded in the FPGA are an integral number of banks wide by some number of rows deep. Each bank has logic blocks for performing functions plus some fixed support hardware. Data can be communicated across banks for functions using more than one bank.

4.4.3.1 The FPGA

FPGADLX's FPGA really consists of two kinds of circuits. One kind is like a normal FPGA and performs the work done by the FPGA. The other kind is the support hardware and is made up of non-logic block devices. Both kinds of circuits are side-by-side in the FPGA, working together to produce results and communicate with the host processor. Perhaps the clearest way to describe the FPGA is to cover each kind of circuit on its own.

4.4.3.1.1 The FPGA Logic

The first thing to notice about the FPGA is that its width is divided into four banks. Each bank can accept two 32-bit operands and produce one 32-bit result. This allows operations on data up to 128 bits across. Since functions can use more than one bank, communication of data between adjacent banks is supported. Likewise, functions occupy some number of rows and integral number of banks. The wide width is intended to allow functions to exploit parallelism in the algorithm being implemented.

Borrowing from the DISC project, each bank is arranged into rows of logic blocks, and two operands are made available to a bank's logic blocks on busses running length-wise through the device. One bit from each of the operand busses is accessible by each logic block. For instance, the fifth logic block in a row can only read the fifth bit from either, or both, of the operand busses. Operand busses drive data for only one clock

cycle and do not buffer data for functions. Therefore, functions that need to have operands stored must latch the operands on the first cycle the function is active and store them internally until needed.

Results produced by functions are placed on a result bus that is separate from the two operand busses. Since there are four banks, there are four result busses. There are several limitations on how results may be driven onto the bus. First, only one function may drive the result bus at a time. Second, access to the bus is granted by the FPGA's support circuitry working in combination with the processor's write back mechanism. Finally, the result coming from a function must appear on single row of the function—most likely, but not necessarily, the last. The second and last restrictions exist because of the features of the FPGA's support circuitry and the way the processor treats results during write back.

Short, local lines are available within the FPGA for functions to move data horizontally along rows and between rows. Since FPGADLX is a PFU-type of reconfigurable computer, data movement and the order of calculation through the FPGA is one-way. This arrangement prevents functions from looping on their own or from reusing their results directly. Instead, function loops are implemented outside of the array, like any other software loop, by testing conditions and then branching based on the result of the test. Functions can resuse results they have produced, but they results must be fed back via the CDB and operand forwarding. This is similar to what has been seen in the

Chimaera and Garp processors. The driving idea here is that space can be saved within the FPGA—and therefore the processor—by eliminating some of the routing found in traditional FPGAs and making use of existing processor features.

To give better definition to other FPGA qualities such as size and operational characteristics, the Xilinx 6264 [45] has been adopted. The 6264 is the largest member of the Xilinx 6200 family of FPGAs. The 6200 series was specially designed to work alongside a microprocessor. The 6200s feature a regular array of fine-grained logic blocks—two one-bit inputs producing a one-bit output—suited to data path designs. 6200s are dynamically reconfigurable; capable of loading netlists in byte, half word, and word increments; and are memory mapped into the processor's address space. Moreover, the address space of the 6200s corresponds to certain positions in the FPGA, thus making it possible to tell what features are located at each address. While FPGADLX's array is not mapped into the host's memory space, this is powerful since it allows a function to be positioned anywhere just by adjusting the addresses where its netlist is loaded. FPGADLX adopts this concept of using coordinates to locate FPGA components. Besides providing computational power, logic blocks have the ability to store a bits and time the movement of data. This allows for the creation of pipelined functions within the FPGA. The 6200s also allow the processor to read and write registers within the array, something not supported in the basic FPGADLX design.

The size of the Xilinx 6264 is 128 by 128 logic blocks. That is wide enough to accommodate the four banks in FPGADLX. The estimated gate count falls between 64,000 and 100,000. Undoubtedly, this FPGA is quite large. Its size becomes more important considering that the FPGA sits on the same die as the host and still requires support circuitry not discussed yet. To bring the size down, but still ensure that the FPGA is deep enough, it is suggested that FPGADLX's array size should be 128 by 64. This half the size of the 6264, which means that FPGADLX should have about 32,000 to 50,000 gates of logic available for functions to use.

The actual layout, design, and logic block capabilities for the FPGA have not been developed because the focus of this thesis is on the overall RC design. Instead, determining the exact features of the FPGA have been left as work for a future effort. Unfortunately, not having a complete design for the FPGA makes it difficult to estimate the performance of FPGA. For performance (latency of functions and so forth) results from other reconfigurable computing efforts are relied. More on this will be discussed in Chapter 6 when dealing with test applications.

Before more of the FPGA logic is described, a distinction needs to be drawn between netlists and FPGA functions. Until now, FPGA functions were thought of as being independent objects. In FPGADLX this is not so. FPGADLX allows functions to interact with one another. It is actually the netlists which are independent. A netlist can contain a single function (which was the assumption up to now) or multiple functions

that cooperate to perform a more complex task than could be accomplished by a single function.

Cooperating functions are possible since netlists are loaded into the FPGA in whole, which allows the functions within the netlist can be set up to interact with one another. To avoid confusion, each function is given a unique numeric identifier by its parent program. The numeric identifier allows the parent program to address each function in a netlist individually which, in turn, gives the program fine control over all the functions in a netlist.

Cooperating functions are appealing for two main reasons. First, it allows for more than two operands to be used, per bank, in a netlist. Because functions can only read from the operand busses on the first cycle they are activated, a function is limited to reading only two operands per bank, or up to eight operands total. However, for some tasks more operands may be required. A netlist with cooperating functions can overcome the operand limitation. This is done by first loading operands into one or more functions and then executing another function which draws on the operands previously loaded into those other functions in the netlist. The second benefit from cooperating functions is that additional assembly instructions and circuitry is not needed in order to load the FPGA with data. Because data can be moved into the FPGA each time a function is invoked, the normal FPGA execute instruction suffices. Section 4.4.4.3 describes the FPGA execute instruction.

In order for cooperating functions to work, the functions must execute is in a certain order. For example, if function 1 is providing operands for use in function 2, function 1 always needs to execute before function 2. Unfortunately, FPGA issue window allows functions to execute out of order. To ensure that functions progress in the proper order data dependencies, either spurious or legitimate, must exist between functions. Because of data dependencies, a function cannot execute until the function producing a needed operand is written back. If the FPGA issue window were changed to in-order issue, then data dependencies would not have to be inserted for cooperating functions with distance zero dependencies (cases in which function 1 is guaranteed to complete before function 2).

It has already been stated that functions, and therefore netlists, can occupy more than one bank and be any number of rows long. What's more, because dynamic reconfiguration is possible, netlists can be located anywhere in the array—with some limitations. FPGADLX allows netlists to be positioned anywhere

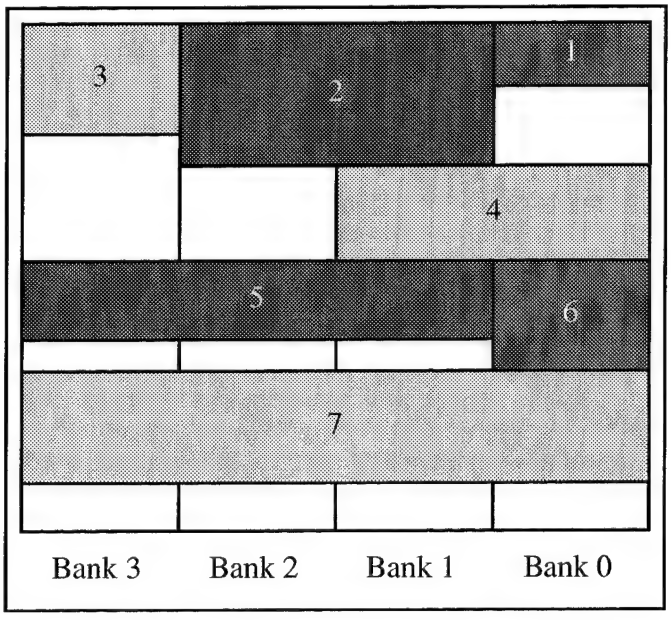

Figure 20. FPGA Loaded With Seven Netlists. This illustrates how the FPGA might look when playing host to seven netlists of various bank widths and row lengths. Notice that empty spaces occur naturally because netlists cannot always be loaded so that they fit together perfectly. Free spaces are also formed when netlists are removed.

there is free space in the FPGA. Netlists that are one bank wide have the most freedom to move since they can load into any of the four banks, provided there are enough unoccupied rows in one of the banks. Double-bank netlists can only go in bank pairs 0-1, 1-2, and 2-3. Similarly, triple-bank netlists are only allowed in banks 0-1-2 and 1-2-3 while quad-bank netlists must use all four banks. Placing multiple bank netlists is more difficult since a vacant space equal to the foot print of the function must be present somewhere in the FPGA. Once loaded, netlists cannot relocate unless removed from the array and reloaded. An example of how the FPGA may look when loaded with several netlists is shown in Figure 20. The job of placing and loading netlists is delegated to the OS.

4.4.3.1.2 FPGA Support Hardware

As might be expected, a couple of things need to be covered before beginning to look at the support hardware. In this instance, the main concern is functions. In FPGADLX, functions are considered to always be active. Or, in other words, functions are clocked and perform work all the time. However, the loading of operands and reading of results is under the control of the host processor. Unlike Chimaera, for example, functions in FPGADLX can be either asynchronous or synchronous (clocked or unclocked) circuits. Functions may have known or unknown latencies. Functions with variable latencies (if-then-else logic, for example) are allowable in a superscalar machine because dependencies between instructions are resolved at run time. Furthermore, since staged circuits are allowed, it is trivial to add the support needed to build pipelined

functions. Both pipelined and non-pipelined functions are possible in FPGADLX. The timing required to compute an answer is solely the responsibility of the function, although the support hardware is used to aid the function in most cases. Functions are also responsible for signaling that they are finished with a task, but not for the writing back of results. Instead, write back is a coordinated effort involving the host processor and the support hardware.

A secondary issue to address is that a special register has been added to FPGADLX. This register stores the process identification (PID) number of the process currently executing on the processor. The PID is a unique identification number assigned by the OS—at least in the UNIX operating system—to each process executing on a computer. The PID register is written to by the OS every time a context switch is performed. The purpose of the PID register will become clear in just a few moments. With this said, the role of the support hardware can now be better understood.

The support hardware is a mixture of active and inactive components which track the state of functions and allow the functions to interact with the host processor. Figure 21 shows how the logic part of FPGADLX's FPGA relates to the support hardware. The seven components comprising the support hardware are found on every row of every bank. Each function has its own support hardware, one row of hardware for each row of the function. Even though functions can cooperate, the support hardware for each function operates independently. Configuring the support hardware for a function is

done when the netlist containing the function is loaded into the FPGA. Like the logic part of the FPGA, the netlist determines the content and behavior of the support hardware. In this manner the support hardware is tailored to match the needs of the function it serves.

Once configured, the support hardware becomes, in essence, a state machine. The state of the hardware can be affected by the FPGA logic, the host processor, and the itself. Just how the seven parts of the support hardware is used and operates is conveyed in the following bullets.

- **Function ID Unit** — The Function ID Unit is a ten-bit register that stores the function number (funcnum) of the function located on a row. The funcnum, a unique number given by the function's parent program, is set by the

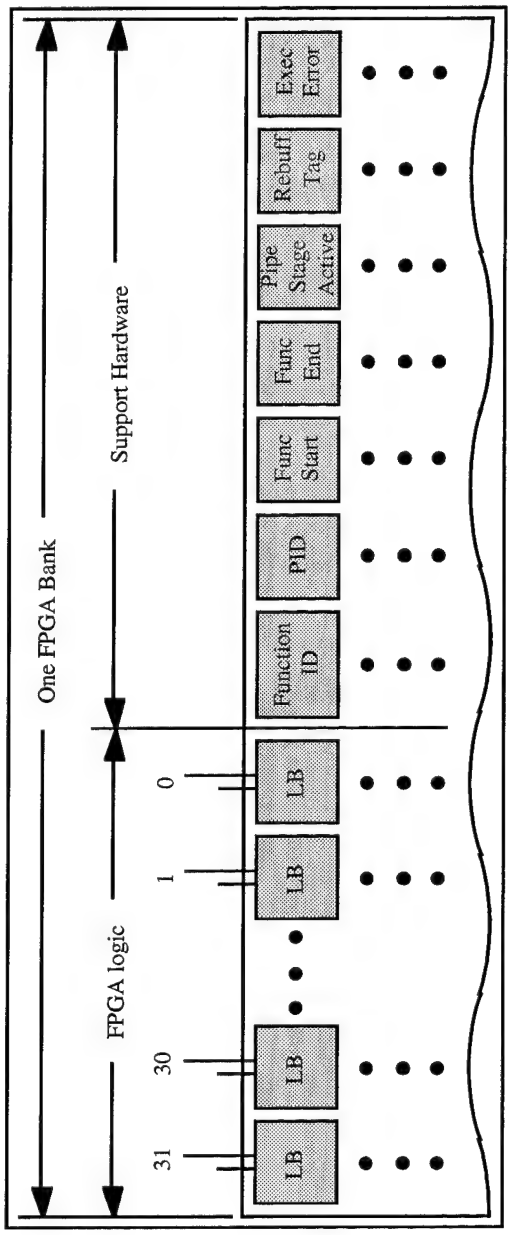

Figure 21. Detail of an FPGA Bank. The bank is split into two parts. One is like a normal FPGA and consists of an array of logic blocks and interconnect. The other part is support hardware and is made up of various registers mainly used for managing and using the FPGA. Some of the registers are set by a function's netlist. Others are under the control of the host processor. This picture is simplified and does not show any intra-bank or inter-bank interconnect, control lines, or busses.

OS when the netlist containing the function is loaded and does not change during the function's lifetime.

- **Process ID (PID) Unit** — The PID Unit is a another register much like function ID. In this case, register stores the PID of the function's parent program (or process). Together, the Function ID and PID units identify a function as belonging to a certain process.

Besides providing ID for a function, Function ID Unit and PID Unit enable the host processor to locate and "activate" functions at the appropriate time. During the clock cycle in which a function starts execution (the clock cycle in which the function's operands are driven onto the operand busses running through the FPGA), the funcnum and PID of the process currently active in the processor are also driven on their own busses into the support portions of the FPGA. The funcnum comes from the FPGA issue window and the PID from FPGADLX's PID register. These values are compared to those stored in the Function ID and PID units of the support hardware. Matching both values causes a true signal to be sent to all logic blocks in the row where the match was made. This signal tells the logic blocks in the row, if so configured, to read data from the operand busses. By including Func ID and PID units on each row of each bank and generating a latch signal, functions are able to read operands at any location within their footprint. This gives the function designer more freedom in building functions.

The Func ID and PID units are also used when writing back results from the FPGA and for aligning operands with the FPGA's banks. Write back of FPGA results is discussed in just a few bullets below. Aligning operands with banks is covered in section 4.4.3.2.

- **Function Start** — Function Start acts as an execution marker for a function when used in conjunction with Result Row, Pipe Stage Active, and Reorder Buffer Tag. A logical true placed in Function Start, a one-bit register, marks one of the function's rows as the row in which Pipe Stage Active and Reorder Buffer Tag should be loaded when a function starts execution. Function Start's value is set by the function's netlist and does not change.

- **Function End** — Like Function Start, Function End is a one-bit marker. In this case, however, it labels the row where Pipe Stage Active and Reorder Buffer Tag should be examined by the host processor's write back logic. The distance in rows between Function End and Function Start is the function's latency—for synchronous functions only. Function End is part of a function's netlist and does not change for the life of the function.

Although Function Start and Function End determine a function's latency, they don't have to be located at on the first and last rows of the function. This is because the timing and latency of a function is a factor of more than just the number of rows making up the function. Some functions may have a latency equal to their number of rows; others won't. For example, it is conceivable to have

functions in which actions take place on separate rows on the same clock cycle. In this instance, functions have a latency less than their number of rows. The possibility also exists that the latency is larger than the number of rows. Because of these complications, the internal timing and latency of a function is determined by the designer when the function is constructed and is only reflected by Function Start and Function End.

- **Pipe Stage Active (PSA)** — As mentioned before, FPGADLX supports three function types: pipelined, non-pipelined, and asynchronous. The PSA hardware allows for the practical use of all three functions. The active PSA units in a function lie between the Function Start and Function End rows of the function, inclusively. As determined by the configuration file, the PSA units in are linked together to form cascaded registers. How the registers behave depend on the type of function they are supporting.

  For pipelined functions, the PSA units take on the value of the unit before them on every clock tick. The PSA in the Function Start row always sets itself to false, unless the function is beginning execution, in which case it is set to true. A function that has any of its first PSAs set to one is unavailable for use. When all PSAs in the row where Function End is set are true, the function is ready with a new result and is a candidate for write back. As long as a last PSA is true, the function is halted (ignores clock signals). When the result is read from one of the function's banks during write back, the last PSA in the bank is set to false and

execution continues. For multiple bank functions, all the PSAs in the last row must be set to false in order for the function to go ahead.

The PSAs in non-pipelined functions behave like their pipelined cousins except that once they receive a true they remain true until reset. With this behavior, the PSAs leave a trail of true values behind as the function they support executes. This ensures that the function stays unavailable for use until it finishes executing. All the PSAs in a bank of a function are reset to false when the result from that bank is written back.

The last PSA unit may be written by the function directly. This capability can be used by asynchronous functions to signal completion. Unlike the synchronous pipelined and non-pipelined cases, asynchronous functions must have extra logic to be able to tell when they have produced a result so that the last PSA may be set at the correct time.

In anticipation of context switching and recovery from incorrect branch predictions, the PSA units can be globally set to false. Done in conjunction with clearing the reorder buffer tag units, this effectively flushes the FPGA of all executing functions.

Reorder Buffer Tag (RBT) — Since the FPGA's banks appear as functional units to the superscalar processor host, it needs to store the reorder buffer entries of the instructions it is executing in the form of a function. This is the purpose of the RBT units. Each unit is a five-bit register because FPGADLX

has 32 positions in the reorder buffer. Like the PSA units, the RBT units between Function Start and Function end are chained together. The chaining is set by the function's net list. The RBTs are also cleared globally during a context switch.

The first RBT is loaded with the reorder buffer entry number of an instruction when a function begins execution. As the function executes, the entry number is passed to the next RBT in the bank on every pulse of the clock. The receiving of the entry number by the last RBT in a function coincides with the last PSA becoming true. The last RBT is checked by the processor's write back logic to match reorder buffer entries with results coming from a function.

Enough material has now been covered to enable a discussion on the writing back of FPGA results. Before jumping right into it, it may be helpful to review how write back is performed in a normal superscalar processor. First, functional units producing a result compete to access one the CDB's busses. (Arbitrating for the CDB is a complicated process and will be ignored for this discussion). Once the FU has secured a bus, it drives its result and the RBT for the result. Issue windows and the reorder buffer snoop on the CDB for RBTs and latch results. Issue windows latch data when a RBT on the CDB matches that of an operand waiting for data (this is how data is forwarded to the issue windows). Entries in the reorder buffer latch results when a RBT matches their entry number. After the FU writes back, it is free work on more computations.

Write back from the FPGA is much like what happens with a normal functional unit. In fact, each bank of the FPGA is treated like any other 32-bit functional unit by the processor's write back and CDB arbitration logic. The only quirk in writing back FPGA instructions is that the location of normal functional units is always known while the position of FPGA functions is not known. This complication means that writing back from the FPGA must also include finding the functions in the FPGA array that are ready for write back. This is where the support hardware helps out. The PID, Function End, PSA, and RBT registers are all used. The PID and Function End units indicate what rows should be checked to see if the function has a result. Since there are four banks, four such searches are performed in parallel. The PSA indicates the status of the function for write back. The RBT gives the reorder entry into which the result should be written. A bank of a function can write back only if the value in its PID Unit matches the value stored in the processor's PID register and the row where Function End is true has a PSA that is true. If write back can occur, a special signal is sent the bank. This "drive results" signal tells the logic blocks in that bank of the function to place the result onto the FPGA's result bus—the result bus is, in turn, connected to one of the CDB's busses (see Figure 19)—and for the bank's last stage to release the RBT onto the CDB. All logic blocks have the ability drive the result bus and do not have to be located in the same row or the last row of the function. However, they must be set to write back by the function's netlist.

Sending the "drive results" signal raises the notion that functions can be somewhat decoupled from their support logic; the signal allows the designer great freedom in building functions since he doesn't have to worry about write back logic and can arrange a function's result to come out on any row, or, even on multiple rows.

- **Execution Error** — The Execution Error is a one-bit register. Functions can use Execution Error to indicate that an error was detected during execution. There are two modes of operation for Execution error.

  The first mode can be used with pipelined, non-pipelined, and asynchronous functions. In this mode any of a function's Execution Error registers can be written, thus allowing error detection to occur anywhere within a function. A logical OR of a bank's error registers is driven onto the result bus during write back. The execution error bits are reset when the function writes back.

  The second mode supports only pipelined functions. In this mode Execution Error registers are cascaded and correspond to each stage in the pipeline, but, unlike other units, the registers don't have to be located between Function Start and Function End. When an error occurs in one stage, the stage's error bit is set and the value steps along with the pipeline. When a result writes back, the last error bit in the pipeline is driven onto the result bus. The last register is endowed with this capability by the function's netlist.

The function designer has responsibility for including error detection in his functions. This involves both the logic required for detection as well as setting the mode and placement of the error registers.

Given the FPGA's limited routing, logic, speed, and reconfigurable nature, the opportunities to catch and flag errors is limited, especially for pipelined functions. This is unlike fixed-logic functional units where detection and flagging is far easier since the algorithms are known in advance and appropriate resources can be brought to bear.

4.4.3.2 The Operand Switch

In section 4.4.2 it was explained that the FPGA instruction window stores a function's operands in instruction order and that each new FPGA instruction sequence starts decoding into an empty entry in the window. By the convention adopted, operands are stored from right-to-left in the window as the instructions in which they are contained are decoded. The window operates in this fashion because it simplifies the decoding of instructions. That is, the window treats every instruction sequence alike and can apply the same logic on each sequence.

As simple and efficient as the FPGA instruction window is, a problem exists, however, in getting operands out of the window and to their correct bank in the FPGA. The problem arises because an entry's operands and the function that uses them may not be aligned. A simple illustration may help to make things clear.

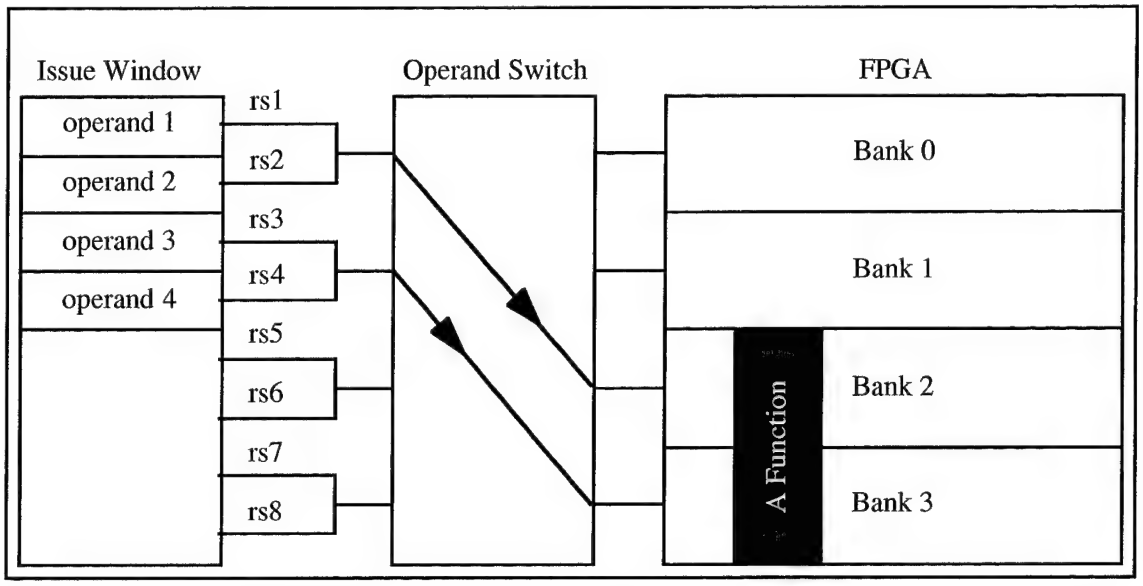

Figure 22. Operand Switch Example. The Operand Switch is shown connecting the FPGA Instruction Window to a two-bank-wide function located across banks two and three. Only the top two switches actually make the connection. The bottom two switches are "don'c cares" and are open circuited.

Figure 22 shows an entry in the FPGA instruction window and the FPGA. The entry contains a fully decoded instruction sequence for a function that is two banks wide. This means that two FPGA instructions were decoded and that the function requires four operands. If the function was located in FPGA banks zero and one, the operands align correctly and no problem exists. In this example however, the function occupies banks two and three. In order to execute the function the operands need to be shifted over to the correct banks in the FPGA. This misalignment between the instruction window's operands and FPGA functions was the reason the Operand Switch was invented.

The Operand Switch is really nothing more than a set of switches that connect operands and reorder buffer tags coming from slots in the instruction window to the operand busses of the proper bank or banks of the FPGA. Individual switches inside the

Switch may either be connected to one of the FPGA's banks or must be open (connected to nothing). Possible connections for the switches are shown in Figure 19 on page 107. The operational rules for the switch follow.

The setting of the switches in the Switch depends on which bank is first used by a function in the FPGA. The FPGA's support hardware allows the Switch to sense the first bank occupied by a function. The Switch need only examine the Function ID units for functions where the PID unit matches the processor's PID register. This can be done by simply ANDing all values involved for each bank. The lowest numbered bank that turns up true is where the function starts. Since there are four banks, there are only four states that the Switch can assume, making operation fairly simple.

1. If the function starts in bank zero, all switches are connected straight across.
2. A function beginning in bank one causes all but the last switch to shift data down by one bank.
3. Starting in bank two moves incoming operands down two banks. The bottom two switches are open. This is the situation captured in Figure 22.
4. A single bank function located in bank three means that all but the first switch are open. The first switch is connected to bank three.

Ironically, a function's bank-width does not play a role in setting the Switch since the reading of operand busses is up to the function itself. All the Switch does is arrange

for operands that might be used by a function get placed on the correct operand busses—thus aligning operands with the correct banks. The Switch does not care which, or even if, operands are used.

4.4.4 Instruction Set Additions

It was unavoidable that the introduction of new hardware to SuperDLX forced the addition of new instructions to the DLX instruction set. Fortunately, the design calls for only four new instructions: one for loading and unloading (deleting) functions to and from the FPGA, one for executing functions, one to write to the new PID register, and one to set the Netlist Loader. In reality, only one new instruction was actually created as three of the instructions are extensions of existing DLX assembly. All instructions make use of existing DLX instruction formats thus ensuring compatibility with existing instruction decode logic. Of course, some amount of decode will have to be modified to so that the appropriate actions are taken for the new instructions, but this would have had to be done for any new instructions.

4.4.4.1 FPGA Load and Unload Instruction

The instruction to load and delete FPGA functions is not a new instruction at all; it is simply a TRAP instruction. In DLX assembly TRAPs are J-type instructions which have the format shown in Figure 23. The procedure for a program performing a TRAP is to put any parameters that need to be passed to the trap handler in registers or on the

stack and then to issue the TRAP instruction. The trap handler will perform its duties and then pass results back to the calling program through registers or the stack. Integer register one (R1) is reserved as a special register for placing of a result code by the handler. A process on FPGADLX follows this protocol when loading or deleting FPGA netlists. In this case, the TRAP transfers control the OS to perform netlist loading and unloading on behalf of a calling program. The OS will handle the trap and return a "good load/unload" or "bad load" result to the program through R1. Details on the interaction between processes and the OS and the actions performed by the OS during function loading and unloading are in section 5.2.1. The loading mechanism and hardware can be found in 4.4.5.

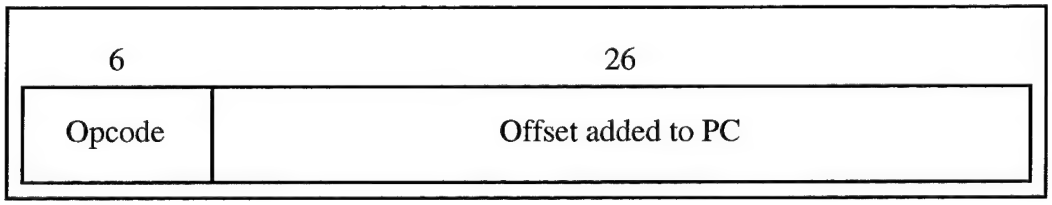

Figure 23. DLX J-type Instruction. Like all DLX instructions the J-type is 32 bits long with a six-bit opcode identifier making up the first six bits. When used as a TRAP, the Opcode field would contain a bit pattern identifying the instruction as containing a TRAP. The remaining 26 bits are an immediate value that is added to the current program counter(PC) to give a destination address where the code for handling the TRAP is located.

4.4.4.2 PID and Netlist Loader Instructions

Similar to the netlist load and unload instruction just described, the PID and Netlist Loader instructions make use of existing DLX instructions and cannot be considered true instruction set additions. In this case the instructions reused are "move an integer register value to a special register" and "move the value in special register to an

integer register" and are identified by the mnemonics MOVI2S and MOVS2I, respectively. Although MOVI2S is used to load the PID and Netlist Loader registers, only the PID register makes use of MOVS2I because nothing is ever read from the Netlist Loader registers. The instructions can be reused because the PID and Netlist Loader registers are special registers and fewer than 32 of the possible special registers allowed by DLX have been defined.

The "move special" instructions use the R-type instruction format shown in Figure 24. Rs1 gives the register number that data is to be moved from. Rs2 is not used. Rd gives the destination register for the move. Actual register numbers have not been assigned to the PID register and the two Netlist Loader registers since no numbers have been given to special registers already defined for DLX. Instead the PID and Netlist Loader registers are identified by the labels PID, NLR1, and NLR2.

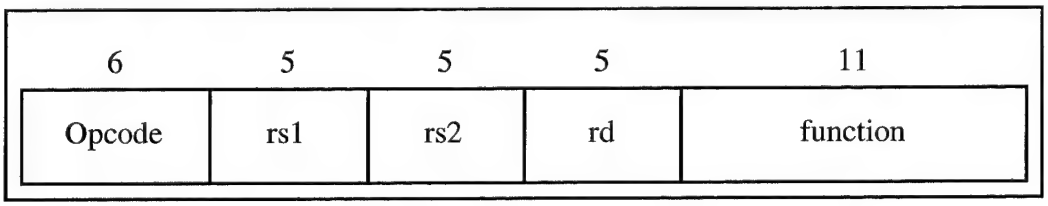

Figure 24. DLX R-type Instruction. The R-type format has space for specifying two source registers, rs1 and rs2, and a destination register, rd. The function field is used to further detail the operation encoded into the Opcode field. For example, all ALU instructions have the same Opcode but rely on the function field to indicate which ALU operation to perform.

Because the PID and Netlist Loader registers are vital to proper system operation, their use must be protected. Therefore, moving data into or out of these registers can

only be done when the processor is in the supervisor mode (i.e. when the OS has control of the computer).

4.4.4.3 FPGA Execute Instruction

The fourth new instruction makes use of the R-type format as seen in Figure 24. While the format remains unchanged, the FPGA execute instruction appears as shown in Figure 25 to the decoder. The function field of the normal R-type instruction has been broken into two parts and is the only change made to the format.

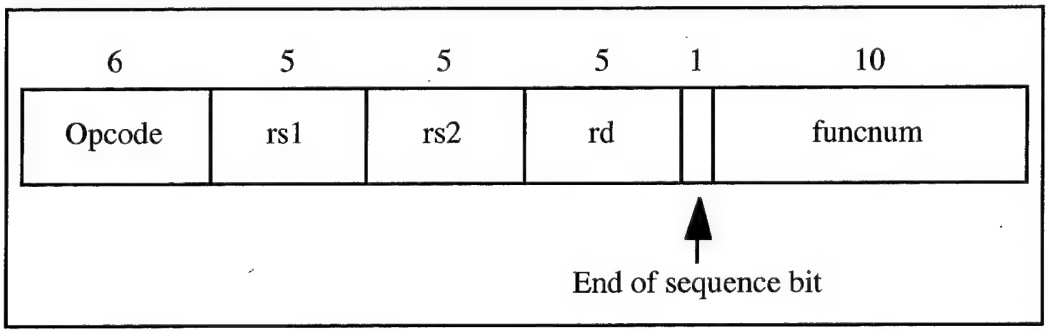

Figure 25. FPGA Execute Instruction. Based in the R-type format the this instruction uses one of the bits from the function field as an end of FPGA instruction sequence indicator. This "stop bit" is needed so the decoder can know when a chain of FPGA instructions has ended since executing an FPGA function can require up to four instructions be decoded. The ten bits remaining of the function field encodes the funcnum used by the program to identify its FPGA functions. The ten-bit field allows each program to have up to 1024 functions active at any one time

The first part is a bit used to mark the end of a sequence of FPGA instructions. This bit is required since the FPGA has four 32-bit banks and it can take up to four instructions to initiate execution of one FPGA function. The bit is examined during decode so that FPGA instructions may be placed into their instruction window properly and to keep different FPGA sequences separated. For instance, if a function to be run

needs two instructions, the first instruction would have a zero for its end of sequence bit and the second would have a one. When the decoder sees the end of sequence bit it knows to begin the decode of any new sequence into the next available entry in the FPGA instruction window. The decoder also protects against code errors by detecting when four FPGA instructions have been seen without an end of sequence bit set in the last instruction and when FPGA function number has changed without an end of sequence having been indicated.

The second part of the function field is ten bits wide. It contains an FPGA function number, funcnum, assigned to the function by the program. Funcnum's size allows for a program to have up to 1024 (2^{10}) FPGA functions active at any one time. Although the FPGA is most likely too small for this many functions in total, the large number insures that there are more available function numbers than possible active functions in a program. Thus, programs will always have a unique funcnum for each of their functions.

The assembly language format for the FPGA execute instruction is:

```
fpga rd,rs1,rs2,function.
```

where rd is the destination register, rs1 is the register containing the first operand, rs2 is the register containing the second operand, and function is the eleven bit field containing the end of sequence bit and funcnum.

The FPGA execute instruction supports only integer registers. Therefore, both the source registers and the destination register for all FPGA functions must be one of the 32 integer registers. The reasoning for this stems from the fact that FPGAs perform poorly on any kind of operation involving floating point numbers. Also, a key point here is that, unlike some DLX arithmetic R-type instructions, immediate values cannot replace the second source register. All operands found in an FPGA execute instruction are interpreted to be register labels. This means that immediate values can be used in an FPGA function only if they have been loaded into a register first. Not allowing immediates as operands is not seen as a drawback since the ranges of the numbers involved is a small subset of possible integer values—0 to 31 for unsigned integers and -16 to 15 for signed numbers. The magnitudes could be increased, but only by consuming part of the eleven-bit field containing the funcnum and end of sequence bit, thus reducing the number of FPGA functions available, and complicating the instruction decode process.

Conversely, the eleven-bit function field *must* be an immediate value; it cannot be a register. Forcing the function field to be an immediate is fine since the value placed in the filed is known at compile time and will not change during program execution.

4.4.4.3.1 Instruction Sequences

References have been made to the existence of FPGA instruction sequences. Instruction sequences are a series of at least one, but no more than four, FPGA execute instructions. All the instructions in a sequence relate to one specific instantiation of an FPGA function. A sequence completely specifies the source registers containing data for a function and destination registers into which a function's results will be placed. Moreover, the instructions are tied directly to banks used by a function. Quite simply, this means that the first instruction specifies the two source registers and the destination register for the first bank occupied by a function. Similarly, the second instruction in a sequence specifies the same but for the function's second bank, and so on. To illustrate the sequence concept, two example sequences are given.

```
fpga r3,r10,r11,#0
fpga r4,r12,r13,#1024
```

Figure 26. A Two Instruction Sequence.

The code fragment in Figure 26 calls on FPGA function number zero (0). Function zero must be two banks wide since the sequence calling it contains two instructions. The eleven-bit function field is represented in decimal notation. When decoded, the function fields for both instructions would be 00000000000 and 10000000000 in binary, respectively. In binary form, it is easy to see that the funcnum is indeed zero and that the second instruction has the end of sequence bit—the left-most bit—set properly to one. In this example, the first bank occupied by function number

zero will have the values of register ten and eleven sent down the two operand busses. Likewise, the values of registers twelve and thirteen will be driven onto the operand busses of the second bank. The first bank will produce a result that is to be stored in register three while the second bank's result goes to register four.

```
fpga r8,r10,r20,#1030
```

Figure 27. A One Instruction Sequence.

The second example (Figure 27) shows a sequence of only one instruction. Converting the function field into binary, 10000000110, reveals that the end of sequence bit is in place and that the funcnum equates to six (6).

The way FPGA results are written back and the manner in which operands are aligned with FPGA banks requires that a sequence must have as many instructions as the number of banks occupied by its corresponding function. However, this does not mean that a function must produce a result from each of its banks or read any operands on a bank. If this were so, cooperating functions, which produce no results, would not be possible. Nor would it be possible to create functions that read in operands from banks where results are not produced.

To meet the custom input and output needs of functions dummy registers can be used where no input operands are needed or results are produced. Actually, any register can be used as a dummy input operand since FPGA functions are directed by their netlist

as to which bits and from what operand busses to read. It matters a great deal, however, which registers are used as destinations. When a bank is producing legitimate results, a true destination register can be used. However, when a bank produces garbage results or no results whatsoever, a dummy result register must be used. For FPGADLX the dummy destination is register zero (R0) since, as defined by the DLX instruction set, it is hardwired to the value zero. R0 is safe to use as a destination register because its value cannot change. Processors without an R0 equivalent will have to designate a register to act as the dummy.

```
fpga r0,r5,r0,#11
fpga r4,r6,r0,#11
fpga r0,r7,r0,#1035
```

Figure 28. A Sequence Using Dummy Registers.

Figure 28 shows how R0 is used in a fictitious code example. The three instruction sequence takes three operands, one for each bank of the function, and produces a single result at the middle bank. R0 is used wherever data is not needed or produced by the function.

The final item of importance regarding sequences is that the instructions in a sequence must occur consecutively; other instructions are not allowed to mingle within a sequence. An illustration of how not to assemble a sequence is shown in Figure 29. The reason for this stems from the idea that blocking instructions in a sequence makes the commit process easier to manage in the case of partially decoded FPGA sequences. (See

section 4.4.6.6 on page 148 for more details.) If instructions in a sequence did not follow one another, the commit logic would become tremendously complex.

```
fpga r10,r5,r18,#112
add r4,r13,r14
fpga r11,r4,r17,#112
mult r6,r13,r15
fpga r12,r6,r16,#1136
```

Figure 29. A Non-consecutive FPGA Instruction Sequence.

4.4.5 The Netlist Loader

The final major piece of hardware added to SuperDLX en route to becoming FPGADLX is the Netlist Loader. As its name implies, the Netlist Loader loads netlists into the FPGA. The Loader is a semi-intelligent device that uses the host's existing memory bus to move netlists from RAM into the configuration memory of the FPGA. Figure 30 shows the Netlist Loader and how it relates to other components of the processor. It can be seen that the Loader consists of two parts: the Configuration Manager and the Memory Arbiter.

4.4.5.1 Configuration Manager

The Configuration Manager (CM) is the Loader's brains. It is responsible for taking information about how a netlist is to be loaded, generating memory addresses and fetching parts of the netlist from memory, and loading those parts into their proper location in the FPGA array. The CM is operated, under the direction of the OS, by loading two registers with data pertaining to the netlist to be loaded. The registers are

accessed using the MOVI2S instruction described in section 4.4.4.2 on page 128. The first row of the FPGA where the netlist will begin to load is specified in the lower 16 bits of Netlist Loader register one (NLR1). The upper 16 bits of NLR1 stores the number of the left-most bank hosting the netlist. NLR2 keeps the address of where the netlist is stored in memory, and acts like a program counter for the Loader. When both registers have been set, regardless of order, the Loader begins loading a new netlist.

NLR1 allows the CM to calculate the exact position within the FPGA where the netlist will begin. Loading begins by reading a bus width's worth of data (128 bits) at the address stored in NLR3. It is assumed that the OS stored the netlist into RAM with a five byte header—the rest of the netlist contains the actual bits defining the

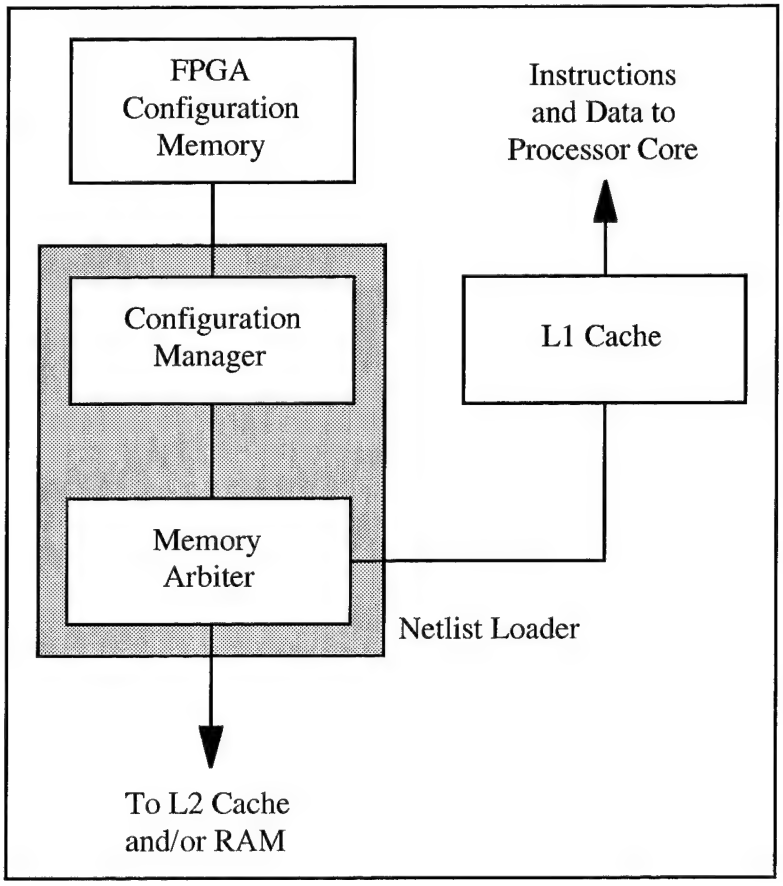

Figure 30. The Netlist Loader. The picture shows how the Netlist Loader relates to the L1 cache and the bus line running off-chip to the L2 cache and memory. The Netlist Loader consists of two devices, the Configuration Manager and the Memory Arbiter. The Manager controls the loading of netlists into the FPGA's configuration memory. The Arbiter regulates access to the bus between the L1 cache and the Configuration Manager.

netlist. The first four bytes (word) of the header gives the length of the netlist (not including the header) in bytes. The fifth byte tells how wide the netlist is in banks (1, 2, 3, or 4). Once the header has been processed, the rest of the netlist is read in cache block chunks and placed into the FPGA array.

The start position and the header information, coupled with the regular, predictable layout of the FPGA array, allows the CM to correctly load the netlist. First, it is assumed that a netlist is arranged so that it specifies an FPGA configuration top to bottom by rows, ignoring any bank boundaries. (It might help to imagine that the netlist is like a folded brochure where the writing continues across the creases instead of staying on a single pages.) Given that features within the FPGA are arranged and addressed by rows and columns—a single logic block or device of the FPGA consisting of multiple columns, it is a simple matter for the CM to place the netlist correctly.

Since the starting point for loading the netlist and the netlist's width in banks is known, netlist placement can proceed by configuring all the columns—with perhaps each column being a byte's worth of data—on a row by row basis until all bytes of the netlist are exhausted. The CM will have to determine which position in the array is being configured at a given time. Within a row this only means increasing the column count. A row has been configured when the number of bytes placed, modulo the number of banks occupied by the function, times the number of columns in a bank, equals zero. When a

row completes, the row count is increased and the column count reset to the leftmost column of the leftmost bank of the function and configuring continues from there.

When the Loader has finished loading a netlist it needs to signal the OS that it is ready to load another netlist. This is done by having the Loader interrupt the processor. Once alerted, the OS begins the loading of the next pending netlist, if any, and sends a "good load" response to the process whose netlist just finished loading.

From Figure 30 it can be seen that the netlists do not pass through the L1 cache on their way from off-chip memory to the FPGA's configuration memory. There are two reasons for this decision. First, the L1 cache is rather small compared to size of a netlist. If netlists were to pass through the L1 cache, the cache would be purged of much, if not all, of its content. Because data has to be saved out of the cache before it can be replaced, causing so many cache misses would potentially add greatly to the time needed to load a netlist. Additionally, the miss penalty once the load finished and normal operation began again, would be increased. The second reason is that netlists, because of the configuration overhead, will most likely not be loaded many times or in close succession, thus negating the effectiveness and purpose of caching them at all. Whether or not netlists should be stored in L2 level is debatable. The impact on an L2 cache would not be nearly as great as it was for the L1, however, the loading frequency still tends to argue against it.

4.4.5.2 Memory Arbiter

As mentioned in the previous section, the Netlist Loader reads in a netlist 128 bits at a time. Part of the reason so many bits are read at once is that the processor already supports data movement of that size to and from the chip. Another part of the reason is that netlists are expected to be rather large, at least in the hundreds of bytes, making large reads fairly efficient. A last incentive is that some of the processor's existing hardware and chip pins can be reused, an idea that is not only convenient, but also cost effective.

The reused hardware in this case is processor's memory bus. Originally, the bus would have been used mainly by the cache for loading and storing data and instructions. Occasionally, the processor would use the bus when communicating with off-chip devices (e.g. UARTS, graphic cards, network interfaces) or when by-passing the L1 cache. In any event, bus traffic appeared to emanate from the L1 cache or terminate there since even processor directed I/O passes through it. Thus, there was only one on-chip contender for the bus. However, with the creation of FPGADLX there is a new contender—the CM portion of the Netlist Loader.

To handle this contention, a device called the Memory Arbiter has been fashioned. The Arbiter has been grouped with the Netlist Loader simply because the loading of netlists led to its creation. The Arbiter's job is to grant off-chip access to either the CM or the L1 cache. Its operation is elementary.

An important issue the Arbiter's operation is which device, the CM or the L1 cache, has a higher priority. Intuitively, the cache has a higher precedence. The reasoning is as follows. It is always desirous to keep the processor as busy as possible doing meaningful work. Because programs are put to sleep by the OS when waiting on a netlist to load, they can do no work. Work can still be done by all the other active processes, but not if the bus is tied up loading netlists. So, to keep processor utilization as high as possible the L1 cache, which serves active processes, needs to have preferred access to the bus. Therefore, the Arbiter gives the bus to the L1 cache on demand. The CM can use the bus whenever the cache doesn't need it. In effect, the CM steals bus cycles from the cache. This arrangement of priorities will tend to slow down the loading of netlists. However, the CM should be able to steal a great number of cycles from the cache and it is expected that the impact on netlist load times should be slight. Since active processes continue to run unaffected by the operation of the CM, this situation seems highly satisfactory.

4.4.6 FPGADLX Superscalar Operation

Previous discussion on FPGADLX centered on the design and operation of the new hardware. Little was said about how that hardware fits into the superscalar operation of the host processor. Yet in order to gain a better appreciation for the entire system, the superscalar aspects of the FPGA and its support hardware must be covered. Here then is the final section on the basic design for FPGADLX.

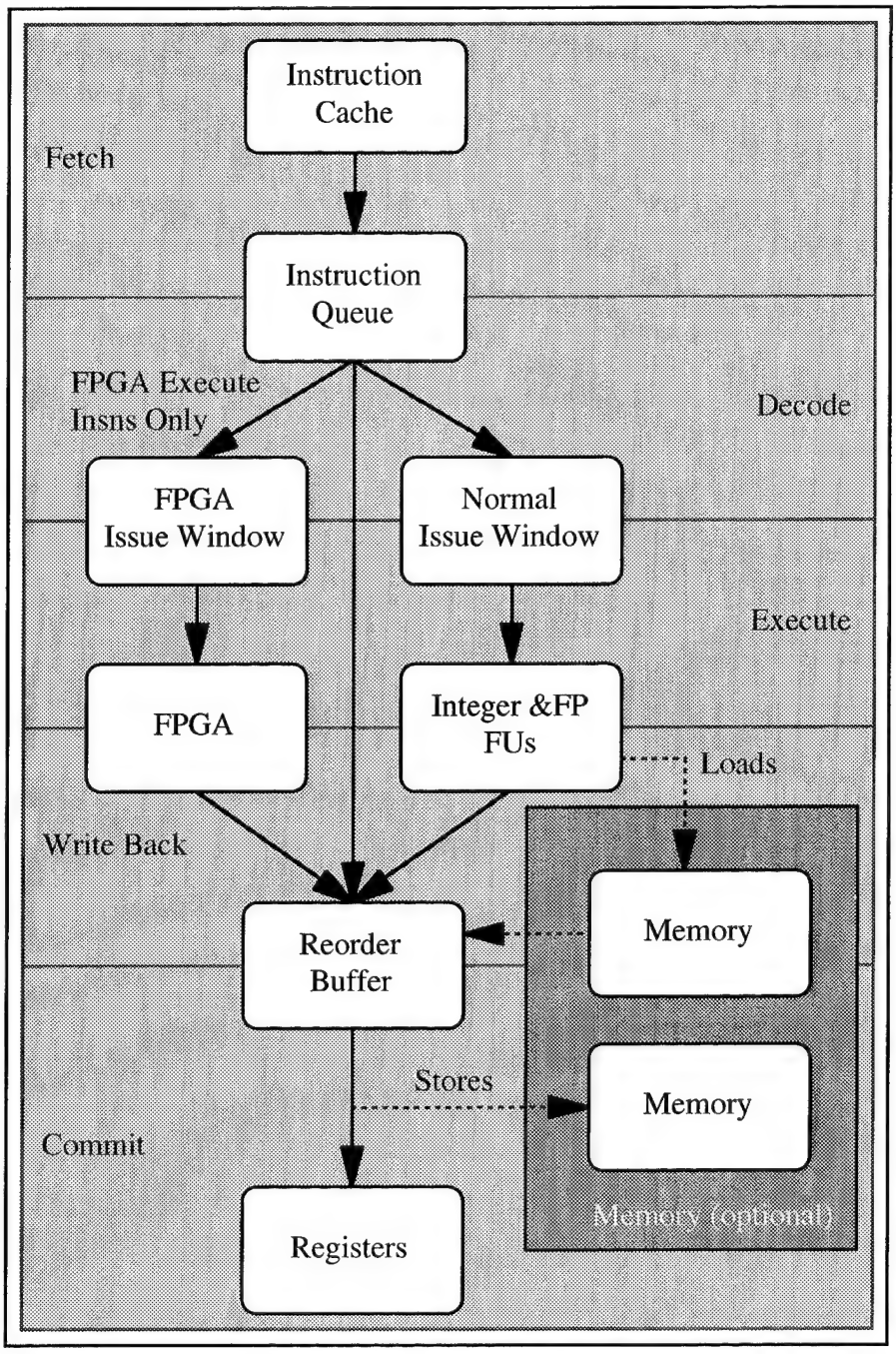

Figure 31. FPGADLX Execution Pipeline. Illustrated is the flow of execution through FPGADLX's forked pipeline. The right hand path is followed by normal instructions while the left hand path is only for FPGA execute instructions. All six stages are shown, including the optional memory stage only used by memory operations. Data and instructions move along the arrows during the stage in which the arrow originates.

Keeping with idea of minimally invasive changes and additions to the operation of the host processor, SuperDLX, the first point to be had is that FPGADLX does not alter the execution pattern of pre-existing instructions. In other words, the same hardware and five pipeline stages (fetch, decode, execute, write back, and commit) remain intact from the SuperDLX design. Additionally, keeping the existing structure of the host processor intact, means that FPGADLX is compatible with all DLX instruction set programs and has the same performance for non-reconfigurable programs as SuperDLX. This, too, was another desirable quality for a viable RC design.

FPGADLX's execution pipeline still has five (six if memory is included) stages and can be thought of as being one entity, but with two paths for instructions to follow as they move from fetch to commit. Figure 31 illustrates this forked nature of the execution pipeline. The right path has the original hardware as described for SuperDLX. The left path uses part of the original hardware, but differs in that instructions move through the FPGA issue window and the FPGA thus constituting the reconfigurable portion of the execution pipeline. Even though the netlist load and unload instruction, the set PID instruction, and the instruction that moves data into the Netlist Loader registers were created because of the FPGA, they do not execute in the reconfigurable part of the processor's pipeline. The only instruction that does use the reconfigurable hardware for execution is the FPGA execute instruction. Since all other instructions use the original pipeline, already described, it makes sense to focus on the FPGA execute instructions and the superscalar operation of the reconfigurable portion of FPGADLX's pipeline from

here on. Actions performed during each pipeline stage are detailed in the next five sections. Two additional sections cover the special case of flushing partially decoded FPGA instruction sequences after a mispredicted branch and handling exceptions raised by FPGA functions.

4.4.6.1 Fetch

The fetch stage, including the branch target buffer and the instruction window, remains unchanged. All instructions are fetched and placed in the instruction window in an identical manner. The handling of jumps and branches is also unaffected.

4.4.6.2 Decode

FPGA execute instructions are still taken from the instruction window and decoded in program order like their normal counterparts, are decoded. The number, a maximum of four of any type, and the overall manner in which instructions are decoded remains unchanged. All instructions are still assigned an entry at the end of the reorder buffer and entered into an issue window.

As might be expected, however, FPGA execute instructions receive special treatment. This stems from the sequence nature of FPGA instructions. Most of this has been covered previously, so only a summary of the actions will be listed here. The first instruction in a sequence is placed in the first slot of an unused entry in the FPGA issue window. The funcnum of the first instruction is placed into the entry's function number

field. While keeping the same entry open, subsequent instructions, if any, in the sequence are placed into the other slots. Operand forwarding and register renaming is performed as instructions enter the issue window. When a sequence has been completely decoded, its entry in the window is marked as full and will move into the execute stage (into the FPGA) once all of its operands become valid. The decoder is responsible for determining when a sequence has been completely decoded and conveying that fact to the FPGA issue window so that decoding of the next sequence begins in a new window entry. Obviously, decoding requires close cooperation between the decode logic and the FPGA issue window.

During decode the FPGA instruction sequences are also checked for errors. If the source code has been compiled correctly, this would not need to be done because all sequences will have be set up correctly. However, correct compilation cannot always be counted on and, thus, the processor must protect against errors. Three types of errors are detected at decode. First is a check to make sure that all instructions in a sequence have the same funcnum. This exposes unexpected changes in funcnums. The second error is when a sequence extends beyond four instructions. The function number field of the active entry in the FPGA issue window is used for detecting these first two errors. The third error when an FPGA instruction that is not the last in a sequence is followed by a non-FPGA instruction. Checking for this error guards against sequences where the instructions do not occur in consecutive order.

4.4.6.3 Execute

Once full, an issue window entry remains occupied until all of the operands to be used by the function associated with the entry are valid and the function itself is ready to accept new data. When these two conditions are met, the entry becomes a candidate for moving into the FPGA for execution. If several entries are ready, the oldest entry has priority.

Only one entry may enter the FPGA on any clock cycle. This seems most practicable, given the complexity of aligning the issue window with functions (the Operand Switch discussed in section 4.4.3.2) and the limited number of operand busses running through the FPGA. Because an entry corresponds to a specific function, this is called "initiating a function." Initiating an FPGA function does not interfere with the issuing of instructions from the normal issue window since the windows are physically separate. Thus, in one clock cycle one FPGA function and up to four normal instructions can issue. Under the right conditions, that could mean that up to eight instructions issue in a cycle.

On issue, the operands and reorder buffer tags stored in the entry are made available to the FPGA and the function in question latches the data coming from the entry. When the function has finished its work and produced results, if any, the write back stage can begin.

4.4.6.4 Write Back

At write back, each bank of the FPGA is treated as an individual functional unit. When a bank of a function in the FPGA has been selected for write back, a "drive result bus" signal is sent to the bank. The "drive result bus" signal causes the selected bank of the function to place its result on the bank's result bus and send its RBT. The result and the RBT values travel as a pair across a channel of the CDB and are latched by entries in the issue windows and the reorder buffer.

4.4.6.5 Commit

For the most part, FPGA instructions are subjected to same the commit process as any other instruction (section 4.3.2.5, page 92). This includes instructions marked as "flushed". The one change to the commit procedure has to do with FPGA instruction sequences during a context switch.

For the purposes of this paper it is assumed that during a context switch the contents of all registers and the PC of the next, non-flushed, instruction in the reorder buffer in line to commit are saved into a process's PCB. This scheme effectively throws away all partially completed work, but provides for precise handling of interrupts and exceptions. One exception to this rule for FPGADLX is that, if the context switch happens in the middle of committing an FPGA instruction sequence, then the sequence is allowed to completely commit before the context switch is honored. If a process were to

context switch out of FPGADLX in the middle of committing a sequence, then the process would resume execution trying to execute the remainder of the sequence. While this partial sequence will execute, it will produce incorrect results since the sequence was not completely specified.

To prevent this from happening, the commit logic must know when FPGA instruction sequence is being retired and when a sequence being retired has ended. This can easily be accomplished by adding two one-bit fields to entries in the reorder buffer: FPGA function and sequence end. Figure 32 shows the new reorder buffer for FPGADLX. The FPGA function field marks an instruction as an FPGA execute instruction. The sequence end field stores a bit to indicate that an FPGA instruction is the last one in a sequence. These two fields would be set during decode. With these two fields the commit logic can delay a context switch until an FPGA instruction sequence is fully committed.

Type 2 bits	Destination 5 bits	Result 64 bits	Valid Bit 1 bit	Flush Bit 1 bit	Error Bit 1 bit	Address 32 bits	FPGA Func. 1bit	Seq. End 1 bit

Figure 32. FPGADLX Reorder Buffer Fields. Shown are the fields present for each entry in the reorder buffer. The fields are the same as for SuperDLX except that two one-bit fields, FPGA function and sequence end have been, added in support of FPGA instruction sequences.

4.4.6.6 Flushing FPGA Instruction Sequences

As mentioned earlier, SuperDLX allows flushed instructions in the reorder buffer to drain out of the processor; flushed instructions execute, write back, and commit

(without storing results in registers). This can be allowed to happen because instructions in the reorder buffer are independent of one another and are guaranteed to execute—each instruction in the issue window enters execution when its operands become valid. For FPGA instructions, this is not the case since instructions appear in sequences. If a completely decoded sequence is flushed, there is no problem. All of the operands for the sequence will eventually become valid, the sequence will execute, write back, and commit like non-FPGA instructions. The real problem arises when a flush occurs while a sequence is only partially decoded.

A partially decoded sequence is both in the FPGA issue window and in the instruction fetch queue at the same time. When a branch is mispredicted and a flush occurs, the fetch queue is cleared, leaving the sequence only somewhat decoded in both the FPGA issue window and the reorder buffer. The is bad for two reasons. First, the FPGA window has an entry that will never close properly. Since the end of sequence bit was never reached, the entry for the partially decoded sequence will remain open until another sequence begins decoding. The new sequence will either cause an error, because of a function number mismatch or an entry overflow, or produce bogus results (if by some chance the new sequence has the same function number and can close the window entry before overflowing). The second reason is that FPGA instructions are expected to drain out of the reorder buffer. Since a partially decoded sequence will never execute and write back, the instructions in the sequence will never be removed from the reorder buffer, and the processor will eventually deadlock. The only thing preventing deadlock would be

if the instruction window signaled an error or if the sequence, by chance, erroneously completed decoding.

Obviously, partially decoded sequences need to be dealt with so that errors or deadlock—both of which shouldn't happen because of a flush—don't occur. Fortunately, the solution is quite easy and already at hand. The first thing to do when a flush occurs is to reset any entry in the FPGA instruction window—there will be at most one—that has a partially decoded sequence in it. The decoder already tracks the decode status of FPGA instruction window entries. Thus, it can reset such an entry easily. This takes care of the instruction window. For the reorder buffer, however, several things need to happen.

The instructions from the partially decoded sequence are marked as "flushed", as would be any instruction. Eventually, these instructions will reach the head of the reorder buffer and will need to be committed. However, they will never commit because their associated function will never execute and write back (the instructions' valid fields in the reorder buffer will never be set). To get around this, the commit logic can be made to detect when it has encountered a partially decoded, flushed sequence and then tricked into removing the sequence from the reorder buffer as if the instructions in the sequence were normal flushed instructions. The commit logic can follow the algorithm given below to detect flushed, yet partially decoded, sequences. Detecting partially decoded, flushed sequences is made easier by the fact that there can be, at most, one such sequence in every

set of flushed instructions and that the sequence must occur at the end of the flushed instructions.

1. Is the instruction at the head of the reorder buffer a flushed FPGA instruction that hasn't written back? Such an entry would have the FPGA function field set and the valid field not set. If no, commit the instruction normally and stay on this step. If yes, go to step 2.

2. Is the entry's sequence end bit set? If yes, go to step 1. In this case, a fully sequence decoded has been detected, but just hasn't finished write back yet. If no, continue to 3.

3. Examine the next entry in the reorder buffer. Is it flushed? If the answer is no, then a partially decoded sequence has been encountered and must be made to commit. To do this, mark as valid all the entries in the partial sequence, from the entry now being examined down to the head of the reorder buffer, and continue at step 1. This will fool the commit logic in step 1 into thinking that the partially decoded sequence wrote back and can thus commit. If the flushed bit was set, then go to step 2.

4.4.6.7 Exceptions in FPGA Instruction Sequences

When an instruction excepts in a superscalar processor, the condition is usually flagged in the instruction's reorder buffer entry and is handled when the instruction attempts to commit to register. When the instruction attempts to commit, the exception

is detected and the processor vectors to an error handling routine—this requires a context switch. This scheme provides for precise exceptions.

Handling exceptions in this way will not work for FPGA instruction sequences since context switches cannot be allowed to occur in the middle of a sequence. Fortunately, getting around this problem is easily done using a combination of the logic suggested in the last section and the commit section (4.4,6.5). This solution does not provide for precise exceptions. However, it does preserve the integrity of the FPGA sequence and ensures that the machine can be restarted.

The first thing to do is to delay the commit of an FPGA sequence until the entire sequence has written back (flushed sequences don't matter). This can be done by detecting when a new sequence has reached the head of the reorder buffer and then freezing commit until the all of the reorder buffer entries for the instructions in the sequence have been marked as valid. If all the instructions in the sequence are valid and none of them have their Error Bit set, then the sequence is allowed to commit. However, if one or more of the instructions have an error, then the processor vectors to the error handling routine at the address of the first instruction in the sequence, without committing any part of the sequence. Fields in the reorder buffer can be used to make the decisions necessary for all this to happen.

4.5 Design Enhancements

The basic RC design that has been developed should provide a good starting point for exploring the merits of reconfigurable superscalar processors. While its details may be somewhat intricate, the overall design and operational concepts are elementary and meet the goals established for a viable design. There are, however, at least four changes or additions to FPGADLX that might make the design even more practical than it already is. The magnitude of impact they might have is not precisely clear, but that they would be useful is evident. It may turn out through experimentation that the enhancements give a tremendous boost to FPGADLX's performance. In any case, most of the additions would add to the already large die size. Thus requiring a tradeoff decision between cost and expected benefit.

The enhancements about to be presented are only concepts. Accordingly, they will be discussed without going into any great detail. Moreover, some of them are featured in other RC proposals, making them less interesting to discuss here.

4.5.1 FPGA Function Caching

Back in section 2.4.1.3.1, the idea of caching FPGA functions in a manner similar to the DPGA concept was presented. The idea was that an FPGA configuration could be saved quickly—in a clock cycle or so—and replaced with another in a kind of FPGA paging. The perceived benefit was that in the space of one FPGA several virtual FPGAs

could be implemented making the FPGA seem larger than it actually is. Adding such a capability to FPGADLX is certainly desirous but would present several difficulties.

To start, having several FPGA pages presents both a management and execution problem. From a management viewpoint, there will be more room for netlists and more locations in which they can be loaded. Thus, the time taken by the OS to fit netlists into the array will almost certainly grow due to the increased complexity. The OS will also have to determine how and when to change pages during a context switch. This will inevitably add to the already time-critical task of context switching.

Considering execution, cached configurations are inactive and therefore do no work. It follows then that all netlists owned by a process be stored on one page. Thus, when a process is active on the processor all of its functions are available for use. When a process is swapped out of the processor it is doing nothing anyway, so it doesn't hurt to have its FPGA page inactive. This reasoning does not preclude several processes from using the same page. It may be technically possible to allow processes to have netlists on more than one page. However, it would probably be very awkward and inefficient considering how data is moved into and out of the FPGA during the execute and write back stages and that processes expect timely execution of functions.

Another inconvenience is that storing pages adds to the complexity of the FPGA and may require extra processor space. Space within the processor could be saved by

building the FPGA in a stacked or layered fashion as proposed in the DPGA papers [17, 15, 16]. But regardless of whether more room is consumed, the cost of fabricating the RC will certainly climb simply due to the added circuitry.

On the plus side, swapping FPGA pages could overlap with context switches especially if all the functions for a process are on the same page. Also, the FPGA's flushing of its pipeline units and reorder buffer units could also happen at the same time.

4.5.2 FPGA Memory Access

Another idea previously mentioned is giving the FPGA the ability to read from and write to memory. This has a great potential to increase the computing power of FPGADLX since it would eliminate, to some degree, the dependence of reconfigurable programs on load and store operations. Coincidentally, the FPGA width matches the processor's memory bus width. As a result, up to four words of data can be stored or read at a time making the FPGA attractive for use on highly regular, packed data problems.

One thing detracting from the perceived benefits is that generating memory addresses requires extra logic in the FPGA. This cannot be avoided. Either functions must calculate addresses or extra hardware must be added to do it. Placing the burden on functions takes away from the already scarce logic resources of the FPGA. Adding hardware consumes die area.

A second drawback is that unchecked memory accesses by the FPGA could result in corrupted memory. Caution must be exercised when building and using functions that alter memory so that data is protected—the FPGA and host must be *synchronized* so that they don't overlap memory accesses to shared data. The current FPGA design has some safeguards built-in. One is that data and execution only moves downward and laterally. This means that a function cannot loop and that all of its memory accesses should be somewhat predictable to a compiler. What other complications may arise are left for future consideration.

4.5.3 Immediates in FPGA Instructions

For the present, FPGA execute instructions must take integer registers as source operands. The reasoning behind this decision was that only small immediates could be supported without changing the format of the R-type instruction. Considering that the setup time for a function—the time a function sits in the FPGA issue window either decoding or waiting on operands, or both, is several clock cycles, giving up on small immediates was an easy choice since immediates can still be loaded into registers with one operation. Keeping the instruction count down, a goal for all programs, led to the adoption of the two input, one output R-type format. With instructions limited to 32 bits, any other format, pre-existing or newly contrived, would have required either more instructions to set up a function or, if more operands were packed into them, would have made instructions inflexible.

All of this notwithstanding, there might be some benefit from including immediates directly into the FPGA instruction. How this is to be achieved in a graceful and sensible manner is a matter for careful deliberation.

4.5.4 Hiding the Cost of Netlist Loads

Currently a process requesting a netlist load is put to sleep by the OS until either it is determined that there is inadequate room for the netlist or the netlist is completely loaded. Putting processes to sleep can effectively hide the cost of loading netlists particularly when there are other processes running on the computer. However, it is not a perfect solution since there are some instances in which it may be advantageous to load netlists while a program remains active and not all operating systems are multi-tasking. Examples include programs that can do useful work while a netlist loads (such as multi-threaded programs) and programs with tight timing constraints.

Under a modified procedure, a process could request a netlist load via a trap, get a "good load" response from the OS immediately, and continue operating while the load takes place. A fairly simple concept, it has one major hurdle to overcome in that processes must be prevented from using functions within the netlist until the netlist has finished loading.

One way around this problem is to implement, in hardware, a table that tracks the status of all functions. Before executing, functions would have to check the table before

entering the FPGA. A function whose entry is not valid won't clear the table and would be prevented from moving from the issue window to the FPGA. Blocked functions will eventually cause the processor to stall, thus maintaining correct program execution order. The current FPGA array has some of the needed hardware already in place for this scheme, however there is no way to check on the operational (loaded or load in progress) status of functions.

A second solution is to protect all sections of code containing FPGA functions with a blocking synchronization command. This synchronization command would be a trap to the OS. The OS would check to see if the netlist specified in the trap has been loaded or not. If the netlist is in place, the process is allowed to proceed past the trap and enter the protected code section. When the netlist is not loaded, the process is put to sleep by the OS and will be woken and when the netlist has loaded.

4.5.5 Function Sharing

With the aim of saving space within the FPGA, netlists and their functions could be shared by many processes. A single netlist could be used by several processes, meaning that copies of a netlist would not have to be loaded. This principle is already employed with FUs; the same FUs are available for use by more than one (actually all) processes. Reuse of FUs can occur because the logic is always present and is generic enough to be used by all programs. Conversely, FPGA functions are transient and are

usually built to satisfy the specific needs of one application. These two characteristics of FPGA functions are the two greatest impediments for function sharing.

If function sharing was implemented, the temporary nature of netlists and functions would present several challenges. The first is to recognize when duplicate netlists have been requested. Given that that can be done, the functions inside the netlist will have to be addressable by each process using the netlist. This requires either more advanced logic than currently found in FPGADLX or that function numbers be assigned by a third party and for use by all processes. Finally, a netlist cannot be removed until all processes using the netlist release them. This would put more responsibility on the OS to track function usage.

The second issue may make it impractical to share functions at all. The idea is that since most functions are tailored to the specific needs of the applications they serve there may be very little duplication of netlists. And, if it is highly improbable that two netlists are ever alike, then it is not profitable to arrange for function sharing.

Intuition says that function sharing is most likely unrealistic. However, it could be employed to spread the cost of loading netlists across many programs. This might make it possible for programs normally unsuited to take advantage of reconfiguration to do so.

V. The OS and Compiler

5.1 Introduction

The previous chapter described the action and organization of the hardware portion of a RC, FPGADLX. However, the complete design of the RC, or any computer for that matter, is not truly complete until the OS and compiler have been addressed. The OS is the active manager of the entire computer system and usually is heavily involved in the execution of most programs. The compiler creates executable code for a computer and must be designed to take full advantage of the computer so that the code is as powerful as possible. For these reasons alone, the OS and compiler for a computer platform are key parts of any successful design.

In this chapter the OS and compiler are looked at in closer detail. Since OS and compiler theory and operation is rather involved, requiring much more and better treatment than can be provided here, the emphasis is on the extra duties that each must perform to support FPGADLX. Therefore, it is assumed that the reader has an a elementary understanding of a compilers and a UNIX-like OS.

5.2 The Operating System

The changes made to SuperDLX on its way to becoming FPGADLX forces a normal OS to acquire two new responsibilities. The first is the loading of netlists into the FPGA array. The second is additions to the actions performed during a context switch.

5.2.1 Netlist Loading and Unloading

For loading and unloading netlists, FPGADLX relies on the OS. There are several reasons. First, the FPGA is a resource that can be shared by all processes. Shared resources need to be controlled in a fair and efficient manner so that all processes have an equal chance at using the resource. Only the OS can guarantee this. Secondly, processes must be prevented from maliciously or accidentally tampering with the FPGA. By allowing only the OS to modify the FPGA, processes cannot corrupt it. Third, FPGA netlists are linked to the process using them and need to be managed along with the process that owns them. Since the OS already controls and maintains processes—usually a process control block or similar data structure is kept on each process—it easily follows that the OS also associate FPGA netlists with their parent process.

As mentioned back in section 4.4.4.1 on page 127, a processes requesting loading or removal of a netlist issues a trap instruction. The trap instruction turns processor control over to the OS at its netlist load/unload handler routine. Before issuing the trap, the process passes, either through registers or via the stack, a set of values to be used by the OS. The first value passed identifies the trap as being either for a load or an unload. The rest of the values differ depending on which action is being requested. For loading, a process passes the name of the FPGA netlist to be loaded (most likely a pointer to a file containing the netlist) and a set of function numbers for the functions in the netlist (the numbers that the process will use when invoking functions). These two items are the

minimal set needed to ensure proper loading and management of the function. For an unload, a process need only pass the netlist name.

A process requesting a load or unload stops executing (is put to sleep) and is not allowed to execute again (woken up) until the request is completed. This means that a process may only request one action at a time and gives the opportunity for a limited amount of FPGA sharing since, while a process is asleep, other processes can request a netlist load. Once the proper values have been relayed and processor control turned over to the OS, the load or unload process can begin. The procedures performed for each are covered in the next two sections.

5.2.1.1 The Load Routine

The load routine of the OS's netlist load/unload trap handler consists of seven steps.

1. The file containing the netlist is opened and its dimensions—rows and banks—are read. The handler's job in this respect can be made easy by providing this information in a header to the actual configuration bits of the netlist.
2. The handler attempts to fit the netlist into the FPGA. Described below is one method which could be employed to great effect.

The OS keeps a map of the FPGA, presumably in some sort of data structure. This mapping indicates which areas of the FPGA are occupied and by what process. The netlist's dimensions are used in a two-dimensional fit algorithm to determine if and where the netlist can load. While finding a spot for the netlist takes a fair amount of computation, it is made somewhat easier by the fact that all functions are rectangular and static. Thus, odd-shapes don't have to be fitted and that netlists don't have to be rotated or rerouted.

If the netlist fits into the FPGA, the netlist is added to the FPGA map and action continues at step 3. When there is no room for a netlist, the requesting function is woken up and the handler returns a "bad load" to it. At this point the handler is finished.

3. The netlist is added to a FIFO queue of netlists waiting to load.

4. The netlist load queue is checked for any entries. If there are pending loads and the Netlist Loader (section 4.4.5) is not busy, the netlist at the head of the queue is removed and begins loading into the FPGA.

5. The first step in the loading procedure is to move the netlist into RAM. In conjunction with the move the OS needs to insert two pieces of data into certain places in the netlist. This data is needed so that once the netlist is loaded, the functions within it will operate properly. The first is the PID of the netlist's owner. The second are the function numbers which were passed to the trap handler. For this to work, it is assumed that the netlist has variables built into it.

The variables would identify where the PID and function numbers need to go. Once the netlist has been moved to memory and set up properly, the next step can begin.

6. The handler sets up the Netlist Loader with the row and bank coordinate of where the netlist is to begin loading and the address of the netlist in memory. This information is transmitted to NLR1 and NLR2 in order. Once the both registers have been set, the Loader takes over and proceeds to load the netlist.

7. Upon completion of a netlist load, the Netlist Loader interrupts the OS. This causes the load/unload handler to activate. The handler arranges for the process whose function just loaded to awaken and returns a "good load" signal via R1 to the process. The handler then loops back to step four.

Returning a "good load" or "bad load" value allows the calling program to take appropriate actions. These actions are discussed in section 5.3.2, page 171, when discussing FPGADLX program compilation.

5.2.1.2 The Unload Routine

Unloading netlists from the array is simpler than loading. In this case the OS simply marks the netlist as deleted in its FPGA map thus freeing its space for use by another netlist. Under proper programming etiquette, the OS returns a "good unload" value via R1. However, doing so is not required since successful deletes are assured—the

function was either removed or was never loaded and therefore did not need to be removed.

Unload is a simple process, but if done improperly can have disastrous results. Chaos can ensue when a process requests an unload and then later tries to use a function in the netlist. If the function has not been overwritten by another, things are fine. However, if an overwrite has occurred, the function will never complete and the process will eventually permanently stall. It is therefore important that unloads are done only when it is known for sure that functions in a netlist will never be needed again, or that if they are needed that the netlist first be reloaded. This determination can be made by the compiler.

5.2.2 Context Switching

For the purposes of this paper it is assumed that during a context switch on a superscalar processor that the contents of all registers and the PC of the next, non-flushed, instruction in the reorder buffer in line to commit are saved into a process's PCB. This scheme effectively throws away all partially completed work, but provides for precise and timely switching. One exception to this rule for FPGADLX is that, if the context switch happens in the middle of committing an FPGA instruction sequence, then the sequence is allowed to completely commit before the context switch is honored (section 4.4.6.5 on page 147 has details). The reorder buffer, issue windows, FUs, and fetch window are then flushed. And finally, the registers and PC of the next ready

process are reinstated and the process begins execution. To this basic routine, there are three simple things that an OS for FPGADLX needs to do to support context switching.

The first is to update the PID register with the PID of the incoming process. The PID register will be needed to locate FPGA functions. An OS may already do this if its processor has such a register.

The second item is to flush the execution status of FPGA functions. Flushing consists of globally clearing the PSA and RBT units of the FPGA. This could be done every time the PID register is reset. The FPGA must be flushed to prevent functions from producing results while their parent process is idle. Of course, having functions execute while their parent process is idle will not interfere with the execution of other processes since functions are identified by PID and function number. However, when the parent process is active again the reorder buffer tags in its FPGA functions most likely won't match with the reorder buffer since it is assumed that the reorder buffer is flushed during a context switch. Any functions left active would either never write back or would eventually corrupt the process with bad results.

The last issue concerns a context switch in which the outgoing process is being terminated. In this situation, the OS must guard against sloppy programming by clearing any netlists from the FPGA that may have not been removed by a process before it quit.

The OS can do this by scanning its FPGA map for netlists owned by the process and then invoking its netlist load/unload handler to remove any leftover netlists.

5.3 The Compiler

The compiler is an essential part of any computer system since it is the means by which programs are turned into executable code. The ability of the compiler to produce efficient and correct code can have a dramatic impact on a program's performance. Today's compilers are remarkable examples of technology. They can take source code written in a high level language and produce binaries optimized to run on a specific platform. To do this requires a tight marriage between compiler and computer architecture. The compiler must have intimate knowledge of the architecture including the instruction set, memory organization, and processor design—number of registers and number and types of FUs).

The relationship between compiler and computer is even more important for a RC. The reasons for this and some of the challenges that lie ahead for RC compilation were discussed in section 2.4.1.1 starting on page 36. The complexities of compiling a reconfigurable program—determining when it is advantageous to use the FPGA, building or selecting netlists and functions, and including functions as part of an executable—are too complex and beyond the scope of this thesis. In fact, compilation is a hot topic in reconfigurable circles and is the subject of intense research. In the interest of brevity in describing a whole system for FPGADLX, it is hereby assumed that reconfigurable

compilation is possible. The intricacies of compilation are put aside for now and left as a future topic.

Instead, for this effort it is more important to look at the tasks a compiler would need to perform for the FPGADLX architecture. This includes how a compiler might go about constructing a binary from facts known about FPGADLX. The discussion about to commence is not intended to cover all aspects of FPGADLX compilation. Rather, it is meant to raise issues, suggest solutions, and show how the compilation might take advantage of some of the novel features of FPGADLX's design. The examination of compilers begins by looking at how netlists might be used and stored in an FPGADLX-based computer system.

5.3.1 Netlist Content and Features

As will be seen in section 5.3.2, netlists need to exist before, during, and after a program is compiled. Netlists must exist prior to a certain phase of compilation so that compilation can be performed intelligently. Then, once the compilation is finished the netlist must remain available for use during the program's execution. The lifetime of netlists implies that they be stored separately from the programs using them. Placing netlists in their own files becomes more attractive when one considers the differences between netlists and an executable binary. While both are just an ordering of bits, a netlist defines a configuration for the FPGA while the binary is instructions interpretable by a CPU. The organization and use of the files is bound to be different.

Merging both types into one file at compile time may be inefficient and inconvenient. There is nothing inappropriate to combining binaries and executables; at least, the operation of FPGADLX doesn't rule out such an arrangement. Indeed, since one must include netlists as part of a compiled program package, the overall size of the program will have same final size whether or not netlists are included as part of a binary or kept in their own files. So, a size penalty involved either way. However, maintaining netlists as separate entities has some advantages. These advantages lead to the design of the netlist loading procedure and the changes to the OS. To help clarify matters and set up discussion on the compiler these advantages are outlined here.

- **Netlists can be changed without recompilation** — Cases may arise in which changing a netlist may provide a benefit. An example would be when a function with increased performance is developed. As long as the changes don't affect the way in which the function is used by a program, a replacement netlist can be built which includes the new netlist. The replacement netlist can be swapped for the one currently in use without recompiling the program.

- **Libraries can be built** — The concept here is the same as with regular software libraries: dependable routines (tools) already exist for doing common jobs. Libraries allow programmers to concentrate on new software and save time by reusing existing software. Having a library of FPGA netlists or functions that could be assembled into a single netlist would be beneficial to RC programmers.

- **Code portability** — The concept of netlist libraries gives rise to making reconfigurable programs portable across platforms and within a RC family. For cross-platform portability, libraries would enable reconfigurable programs to be compiled easily and quickly on any platform since the libraries contain FPGA netlists pre-developed for each system. Intra-family portability would allow programs to adapt to varying FPGA designs within the same instruction set architecture without recompilation simply by changing netlists. This would allow the FPGA's capabilities to be upgraded over time without making reconfigurable programs obsolete.

The concept of netlist libraries is interesting and worthy of expanded discussion. In software libraries, the routines in a library are purposefully written to be used by as many programs as possible. When used, the routines are adapted to fit the program they serve. Just the opposite would be true for a netlist library. The reason can is that netlists represent physical circuits that cannot change. Thus, a program using a netlist must adapt to the netlist. This "reverse accommodation" requires that a compiler know certain qualities about the netlists it will use. To meet this and other needs—to be covered in the next section—for knowledge, the netlist profile was created.

The profile tells the compiler certain information about the netlist. For instance, to use the netlist correctly the compiler needs to know how many functions are in the netlist, what each function does, the latency of each function, how functions are related

(cooperating functions), the quantity and location (which bank) of input operands for each function, and the number and location of meaningful results output by each function. The profile can be included as part of the netlist file or stored separately but while remaining linked to the netlist.

5.3.2 Program Construction

With netlists squared away, attention can now shift to how a compiler might go about making a reconfigurable executable.

The examination of program construction and examination begins at the point in compilation when it has been determined that a certain section of code is a candidate for running in FPGADLX's FPGA. How this determination is made is beyond the scope of this thesis, but for the near future one can easily envision the use of compiler directives inserted by the programmer around key pieces of code. As compilers become more advanced, they may be able to identify candidate sections of code on their own. Nominating code to run in the FPGA is a big step in compilation, but it doesn't mean that the code will necessarily be replaced with an FPGA function or functions. Code should only be replaced if execution in the FPGA is expected to overcome the overhead involved—the time penalty of loading of a netlist versus the time saved by executing in the FPGA. For convenience, this will be called "netlist worthiness."

To figure the netlist worthiness, the compiler must know three things. First of all the compiler must be able to determine, or could be told, the number of times a candidate piece of code is going to execute. This valuation may be difficult, or even impossible, to resolve depending on the nature of the program. The second is the number of clock cycles saved by executing one pass through a section of code, software cycles, in the FPGA, FPGA cycles. The number of cycles taken in software can be calculated by looking at the assembly generated during normal compilation. The FPGA cycles can be taken directly from the netlist or calculated. The third bit of knowledge is the length of the netlist in bytes. The netlist's length enables the compiler to figure the number of clock cycles needed to load the netlist. In FPGADLX's case, this number of cycles is, at a minimum:

$$\left\lceil \frac{NLS}{16} \right\rceil \cdot ML$$

Where NLS is the netlist size in bytes and ML is the latency of a four-word (size of a cache block) read from RAM. This is assuming that memory transactions are not split and that memory is not banked.

Using these three pieces of information netlist worthiness can be expressed as an inequality:

(time to execute a pass in FPGA * number of passes) + netlist load time

≤ (time to execute a pass in software * number of passes).

If the left side of the equation is less than the right side, then the netlist is worthwhile. Undoubtedly, there will be times when the netlist worthiness just can't be determined even by the best efforts of a compiler or a human. In these situations the compiler might have to resort to a crude estimate or depend on a human's judgment.

Once the decision has been made to replace a certain section of code with FPGA functions, the functions can be integrated into the code. The first thing to be done is to request a netlist load. For this the compiler must arrange to have some parameters placed in registers or moved to the stack and then issue a TRAP instruction that calls the OS's netlist load/unload routine. The parameters to be passed were covered in section 5.2.1 on page 161. In order to pass parameters, the compiler must know the name of the netlist to be loaded and have profiled the netlist. By "profiled" it is meant that the compiler has examined the netlist and knows how to use the function or functions within it to replace the code section. At this time the compiler needs to know the number of functions in the netlist so that it can assign numerical identifiers to each function. The identifiers will be used to invoke the functions when they are finally used.

The compiler is allowed to number its functions since it is felt that this eases the compiler's job. There are two main ideas driving this bit of intuition. First, because the compiler has full vision over a program, it can enforce the rule that all functions in use at a given time have unique identifiers. Second, the compiler assigns numbers at compile time which eliminates the need to store function numbers in registers or memory in order to

invoke functions. Both notions have the combined effect of placing responsibility for proper program execution in the hands of the compiler. This reduces the chances that a third party, the OS for example, can interfere and cause a program to execute improperly. The two ideas also save time during execution since everything is set during compilation instead of being handled dynamically once the program is running.

Continuing with compilation, the TRAP instruction returns a value to the program through R1. This value tells the program that the netlist load was either success or a failure. Returning a "good load" tells the program that its netlist is ready to be used. When a "bad load" is received, the compiler has four options for dealing with it.

1. It can loop through the netlist load procedure until successful—nothing prevents a process from asking repeatedly.

2. It can try to load different netlists. It is in a program's best interest to use the best function or functions possible. Unfortunately, FPGA space constraints may prevent the most preferred netlist from loading—larger functions are usually more powerful. However, some code sections may still benefit from using weaker functions.

3. It can forget about running that piece of code the FPGA at all and fallback to using the software version (normal code). This should probably be a permanent option since it guarantees that a program will run if the FPGA is full or if there is no

FPGA at all. This is a reasonable option since compilers already know how to generate non-reconfigurable binaries.

4. Terminate the program.

The first three options open up new possibilities for reconfigurable programs because the compiler has the ability to recover and redirect how the program will execute. In a way this is good in that a program can run without depending on the FPGA or can still achieve speedup by running less potent functions. The main drawback is that there are more options to consider, and perhaps implement, thus making compilation tougher and increasing the overhead for using the FPGA, in the case of fallback functions. Another negative result is that programs and the code compiled from them grow in size. In any event, the compiler must ensure that arrangements are made for loading netlists and dealing with the result of the load, good or bad.

The next step in compilation is to generate the FPGA execute instructions that will replace the candidate section of code. Here, again, the profile information on a netlist is used so that the compiler may issue instructions and thus invoke functions in the right order. For example, the compiler must know about cooperating functions so that it can schedule them in the right order by ensuring that data dependencies are present between them. The profile also tells the compiler where to place input operands and expect results to come out an FPGA instruction sequence.

The final job for the compiler is to unload FPGA netlists when no longer needed. The act of unloading is elementary; the compiler issues a TRAP to the OS's netlist load/unload handler with a request to unload a certain netlist. Not so easy is determining when it is safe to unload a netlist.

Basically, there are two situations under which netlists can be discarded. The first is when it is known for sure that the netlist will never be used again in the program's lifetime. The second is when there is some advantage to be gained by removing the netlist temporarily and the overhead of reloading the netlist can be tolerated. Such a condition might be when the netlist won't be used again for a long time and the space freed by the removal could be used by another netlist. Determining when one of the two situations has occurred is the difficult part. As a simple solution, a compiler could simply unload all netlists right before the program terminates. This inelegant, but effective, method will work even when some of the netlists were never loaded since the OS ignores unload requests for absent netlists.

5.3.3 When to Load Netlists

A last issue for a compiler is deciding when to load netlists during the course program's execution. Some aspects of load timing were touched on lightly in the previous section. Load timing is discussed here because it impacts not only the performance of a single process, but on all reconfigurable processes running at any given time. Basically,

there are two options for a program: loading netlists up front and loading as needed. The pros and cons of each alternative are covered below.

Loading all netlists, or a significant number of the netlists, needed by a program at the start of execution can provide several benefits for the program in question. Chief among them is that it creates the illusion of faster execution time, at least to humans. The mirage develops because loading is done during program startup and is partially hidden by the overhead of starting up. Another plus for up front loading is that a program has a chance to claim much of the FPGA for its own use, especially if the program is the first to use the FPGA. On the flip side, the FPGA may be heavily occupied which means that all netlists cannot be loaded at once. A second drawback is that up front loading by a single process has the potential to limit the FPGA access of all other processes to the detriment of the other processes' performance.

Scattering netlist loads throughout a program so that loading is done as needed relieves demand on the FPGA and increases the odds that the FPGA can be shared by multiple processes. This is even more true if netlists are unloaded as soon as they are no longer required. However, better sharing odds don't necessarily mean that the FPGA will be any less full at any given time. This style of loading may cause programs to stall in the middle of execution while waiting to load a netlist.

Comparing the two load styles leads one to conclude that the best performance occurs with up front loading, but only at the impairment of other processes. Neither one can help guarantee a process's access to the FPGA, although scattering loads may allow more processes to share the FPGA at once than the up front method. Which style achieves the best results for FPGADLX cannot be immediately determined. It may take some amount of experimentation in order to decide. Of course, there may be other load timing schemes which work better than the two describe here.

VI. FPGADLX Simulation Results

6.1 Introduction

Chapters 4 and 5 covered the design and operation of a hypothetical superscalar reconfigurable computer system. The system includes a processor, FPGADLX, and changes to a typical OS and compiler to support the processor. One key factor in the success of the RC system is performance. Consequently, performance was given great consideration during the design's creation. Yet it is hard to determine the performance of the design without being able to test it, especially when a dynamically scheduled processor is involved. For this reason, a software simulation of FPGADLX has been constructed.

The simulator emulates the operation of the FPGADLX processor and provides a preliminary, conservative estimate of how well an actual processor might perform. FPGADLX's advantages over the SuperDLX host processor are revealed by running reconfigurable and non-reconfigurable programs and observing the reduction in the number of clock cycles required for execution. The simulator is not fully operational; it does not emulate all aspects of the RC design. However, it does serve as a useful tool for experimenting with various design decisions and gives a rough estimation of FPGADLX's capabilities and advantages over plain superscalar processors.

This chapter explores FPGADLX's performance by looking at several test programs. The test code was compiled from short C programs that implement the kernels of applications expected to benefit from running in an FPGA. Other programs were also tested with the goal of exploring alternative uses of FPGADLX. In order to better understand the simulator and the accuracy and meaning of the test results, some background on the simulator is given.

6.2 The Simulator

The FPGADLX simulator has most of the features of FPGADLX as described in Chapter 4. Accordingly, it has performance characteristics similar to what a fully functional simulator might have. Adding the missing features has been left for the future as refinements and improvements to the FPGADLX design are made. In depth understanding of the FPGADLX simulator is not important for the purposes of this chapter, but for more interested readers, it is detailed in Appendix A.

The simulator was constructed in much the same manner as FPGADLX was. A simulation of the host superscalar processor was built and refined and then the reconfigurable parts of the design were added. In this case a superscalar DLX simulator was selected and transformed into the FPGADLX simulator. The DLX simulator chosen was SuperDLX, the result of a master's thesis by Cecile Moura while a student at McGill University in Montreal, Canada. [41] SuperDLX is a highly modified version of the original DLX simulator developed as a teaching aid for Hennessey and Patterson's

Computer Architecture: A Quantitative Approach series of textbooks which first introduced the DLX instruction set. [27] The Moura SuperDLX program had several shortcomings which made it unrepresentative of typical modern superscalar processors. As a result, the program had to be partly reworked. The updated simulator is faithful to the SuperDLX design as presented in section 4.3.2 beginning on page 83.

To make an FPGADLX simulator, several parts of the FPGADLX design were added to the revamped SuperDLX simulator. The FPGADLX simulator includes the FPGA instruction window, the logic to decode FPGA instructions and handle sequences, and the FPGA array. The instruction window and decode logic have been implemented true to their descriptions from Chapter 4.

The simulator does not include the Netlist Loader or the Operand Switch because they were not critical to estimating the performance of FPGADLX. Thus, the netlist load/unload trap instruction and the modeling of the load and unload process are also not supported, although, the cost of loading netlists can still be estimated and factored into any performance estimates. Since there is no loading and unloading, functions in the FPGA are considered to be permanently loaded. In actuality, functions are compiled into the simulator in anticipation of being used.

The FPGA array operates as described in Chapter 4 and is fully functional except for a few things. First, the FPGA array does not support multiprocessing. It was not

included because the SuperDLX simulator doesn't accommodate it and an OS for SuperDLX doesn't exist. Second, functions are allowed to be pipelined or non-pipelined but cannot be asynchronous—have variable latencies. However, asynchronous functions could be supported with some extra work to the simulator. The third exception is that the FPGA is considered to be of infinite size. It still has four banks, but instead of 64 rows there is no limit. The infinite size is a result of not loading netlists as part of the simulation.

One nice feature about the FPGADLX simulator—and one of the reasons that Moura's SuperDLX simulator was selected as a building block—is that it operates on assembly instead of binary code. This makes it extremely convenient to convert compiled test programs into their reconfigurable equivalents. Another attractive attribute is that the simulator gives detailed information about the state of the processor while executing. Once a program has terminated, a comprehensive list of statistics about the performance of the processor can be had. The two most important statistics would have to be the number of clock cycles needed to execute a program and the dynamic instruction count.

A drawback to the simulator is that it lacks an OS. Thus, FPGADLX cannot be tested under multi-user, multi-tasking conditions. Another minus is that the simulator offers support for only a few types of I/O functions. This tends to limit the capability of test programs. However, it is not terribly critical since most of the test programs are just the kernels of larger programs and work-arounds to any I/O limitations can usually be

found. A final shortcoming is that the simulator provides only one level of memory; there is no memory hierarchy. Instead, a perfect cache with a one cycle latency is assumed. This robs the simulator of some of its real-world accuracy. However, it could be considered beneficial for the testing done in this effort since FPGADLX's performance is a function the processor itself and is not influenced by the action of the cache or the organization of data stored in memory.

6.3 Test Methods

The creation and execution of test programs on the FPGADLX simulator was relatively straightforward. Each test consisted of running both a software and reconfigurable versions of a program. The goal was to obtain the number of clock cycles needed to complete each version of the program so that a comparison of reconfigurable execution to normal execution might be made. Running both versions through the simulator does not compromise the results since the FPGA hardware does not interfere with the operation of software-only programs. The procedures used to prepare programs for testing and the conditions for testing are given in the next two sections.

6.3.1 Test Procedures

The procedure begins by selecting a program for testing. There was no set criterion for selecting test programs, barring that the programs demonstrate the use of the FPGA in some sort of computing task. Although, one loose requirement was that the

programs selected show the FPGA being used in a variety of ways. Accordingly, the FPGA functions used in the tests range from the very simple to the highly complex.

Once selected, a test program was compiled and run on a SPARC workstation to check that it operates correctly and for debugging. The same C source code is then compiled into DLX assembly and run on the FPGADLX simulator. The results produced by the program on the simulator are compared with the SPARC results to make sure that the simulator works correctly for the software-only version of the program. The test program's DLX assembly is copied and modified to make a reconfigurable version of the program. Modifying the program consists of replacing the some of the assembly instructions with FPGA execute instructions. At the same time, FPGA functions are added to the simulator to support the reconfigurable program now being created. After all the changes are made, the reconfigurable version of the program is run on the simulator and its results are checked against the software and SPARC trials.

If the results from all three runs show that the program is executing correctly, the testing of programs went in one of two directions based on the size of and nature of the programs. For the smaller and simpler programs extraneous print statements were removed from the C source, leaving a bare-boned version of the test program. The stripped program contained only the essentials needed for execution, thereby ensuring that, when compiled, the software and reconfigurable versions of the program were pure function. This enables a better comparison of the speed of the software and

reconfigurable versions. The stripped program is then recompiled into DLX assembly, run on the simulator, and the number of clock cycles expended and dynamic instruction count recorded. Likewise, a reconfigurable assembly file is created, run on the simulator, and performance data noted. The speedup of the reconfigurable program, relative to the software program, can be obtained by dividing the software results by the reconfigurable results.

For larger programs, such as the IDEA encryption algorithm and the discrete cosine transform, where creating a reconfigurable version took a great deal of time, a different approach was used. These test programs were constructed with the computational kernels—the parts to be replaced by FPGA execute instructions—surrounded by `#ifdef` and `#endif` compiler directives. These directives made it possible to control the compilation of the kernels. From the source code, two versions of the software program were created, one being the entire test program and the other consisting of everything but the code kernel. Subtracting the number of clock cycles executed for the second version from the number taken for the entire program leaves the number of cycles required to execute the kernel—with the kernel being the item of interest. The reconfigurable version of the test program was created from the assembly produced by compiling the entire test program. The number of clock cycles required for the reconfigurable version minus the number of cycles for the non-kernel software version gives the number of cycles used by the FPGA to execute the kernel. Dividing the clock

cycles needed to execute the software kernel by the number of cycles for the reconfigurable kernel gives the speedup of the reconfigurable kernel.

To achieve the best results and maintain consistency, all programs run on the simulator were compiled with the fullest optimization possible. Branch prediction was also turned on in the simulator as it consistently decreased the number of clock cycles required for programs.

6.3.2 Test Conditions

Besides the test procedure itself, there are several other issues which affect the performance of the simulation. These factors are described here so that their impact may be understood.

The first issue deals with the modeling the characteristics of FPGA functions—size and latency. The ideal situation would have been to build actual netlists for Xilinx 6200 family chips. The latency of functions created this way would have been fairly accurate since the logic parts of FPGADLX's FPGA are modeled on the 6200 family. Unfortunately, the tools available could only program older Xilinx 4000 series FPGAs. Also, it would have taken several months to complete the netlists for the test programs. Instead, functions implemented on the Garp FPGA array have been adopted. Using Garp as a reference is actually quite suitable since the capabilities and features of the Garp array are similar to FPGADLX's. [26] There are two papers on Garp which mention the

latencies and sizes (FPGA area consumed) of various functions. These papers will be referenced when appropriate.

The second topic concerns the DLX compiler that produces the assembly files that run on the simulator. The compiler is GNU v2.0 modified to produce DLX assembly. The compiler has two shortcomings that affect the performance of FPGADLX. The first is that it only optimizes code to the -O level—the lowest level of optimization available with the GNU compiler. This is unfortunate because the compiler may not be producing the best assembly possible. Fortunately, code compiled with optimization performs significantly better on FPGADLX than non-optimized assembly, all the more so when branch prediction is active in the simulator. The second negative issue is that the compiler doesn't schedule code with a superscalar machine in mind. As a result, the assembly is sometimes not as fit for FPGADLX as it could be. Even so, the temptation to reschedule compiler generated instructions by hand was avoided. All code run on the simulator is unchanged from the way the compiler generated it, except for the modifications required to convert portions of programs to run in the FPGA.

6.4 Test Programs and Performance

Several test programs were built and tested on FPGA in order to compare a normal superscalar processor to a FPGADLX. To put each test program in perspective, a short explanation of its purpose and usefulness is given. Following that, the results of each test and vital statistics about the program are presented. The test programs are

covered in order from the least to the most complex. The source code, software assembly, and reconfigurable assembly are included in Appendix B.

6.4.1 Program One: Bit Reversal of 10-bit Integers

Reversing the bits of an index for an item in an array or list so that the content of the item can be swapped with its bit-reversed mate is a common practice in many signal processing algorithms. Speeding up the process would benefit a great number of applications which currently rely on either awkward software routines or pre-computed tables—both of which consume many clock cycles. Fortunately, reversing bits in hardware can be done without any calculation simply by connecting the input to the output in a reversed order. An FPGA should be able to do this quickly.

This program bit reverses the addresses of a 1024-item array. At 1024, bits only the least significant ten bits need to be reversed to find the corresponding mate. The program actually reverses all 1024 items, even though some addresses are their own reverse. The program uses a simple doubly-nested loop routine (shown in Figure 33) to perform the reversals. The outer loop selects increments the number to be bit reversed—from 0 to 1023. The inner loop, consisting of a number of shifts, builds each of the bit reversed answers in ten passes. This version of the program has a run-time on the order of $n\log_2(n)$. Where n is the number of items in the array. The entire inner loop is executed in one bank of FPGA in the reconfigurable version of the program. The FPGA's execution time also depends on the number of bits to be reversed, but is not directly tied

to the number of bits as was the inner loop of the software routine. It is safe to say that the reconfigurable routine is nearly linear—on the order of *n*.

```
#define SIZE 1024

for (j=0; j<SIZE; ++j)
  {
    input = j;
    output = 0;

    for (i=0; i<10; i++)
    {
      output <<= 1;
      output = (input & 0x1) | output;
      input >>= 1;
    }

    array[j] = output;
  }
```

Figure 33. Bit Reversal Code. The main part of the program consists of a doubly-nested loop. The outer loop increases the number to be reversed. The inner loop does the actual reversing.

The software assembly takes 40,994 cycles to run. With either a pipelined or non-pipelined, one-bank wide reversal function, the reconfigurable program can sustain a minimal cycle count of 5,667 for function latencies of six (6) cycles or less. This equates to a 7.24 times speedup for the reconfigurable version of the program. If the bit reversal program were rewritten so that four reversals could be performed in parallel and a four bank wide FPGA function was used, the reconfigurable program would nearly quadruple its performance, thus executing at most 30 times faster than the software version.

6.4.2 Program Two: Gammatone Filter

Program 2 executes a quick version of the Gammatone Filter [47] using 16-bit integers instead of floating-point numbers. The Gammatone Filter is an accelerated four-stage digital filter. Normally, the Gammatone Filter is used to breakdown sampled sounds at various frequencies along the aural spectrum. A typical Gammatone program would consist of a number of filters each executing in rapid succession to analyze sampled data at close to real-time rates. This particular implementation of the filter was developed by AFIT student Sam SanGregory as part of his doctoral research.

The algorithm used in the test program performs the four stages of the filter in a pipelined fashion, as shown in Figure 34. Thus it takes four passes through the filter before the effects of an input value is felt. This pipelined algorithm essentially unrolls four passes through the filter into one loop, thus exposing some parallelism inherent in the basic Gammatone algorithm. This is ideal for implementing in FPGADLX's FPGA since each of the four filter stages can be performed in parallel by placing one stage in a bank of the FPGA.

The test program used here implements a Gammatone Filter operating at 1KHz. Making the filter operate at 1KHz is done by setting three variables to appropriate values. The test program could actually implement the filter at any frequency just by changing the variables (represented by ap1, ap2, and ap3 in Figure 34). The program begins by sending a noise spike—an impulse function—to the filter. The program then

loops 512 times with no input—actually a zero (0)—being sent to the filter. The loops allow the filter to respond to the impulse function, resulting in a slowly decaying sinusoid. As mentioned before, the program uses only 16-bit integer values.

```
output_store =((X[0] * ap1) + (ap2 * Y[0]) + (ap3 * Y[1]))>> 15;
X[0] = data;
Y[1] = Y[0];
Y[0] = output_store;

output_store =((X[1] * ap1) + (ap2 * Y[2]) + (ap3 * Y[3]))>> 15;
X[1] = Y[0];
Y[3] = Y[2];
Y[2] = output_store;

output_store =((X[2] * ap1) + (ap2 * Y[4]) + (ap3 * Y[5]))>> 15;
X[2] = Y[2];
Y[5] = Y[4];
Y[4] = output_store;

output_store =((X[3] * ap1) + (ap2 * Y[6]) + (ap3 * Y[7]))>> 15;
X[3] = Y[4];
Y[7] = Y[6];
Y[6] = output_store;
```

Figure 34. Gammatone Filter Loop Contents. The computation loop of the Gammatone Filter test program consists of four filter stages unrolled to increase parallelism in inherent in the algorithm. All variables are 16-bit integers. The values Y[0], Y[2], Y[4], and Y[6] are carried over as inputs for the next pass through the loop.

In the reconfigurable version of the program each pass through the filter is performed entirely inside the FPGA, with each bank of the FPGA performing one of the four filter stages. The four instructions making up the sequence for the FPGA function contain: the three variables determining the frequency of the filter, the new input value for the pass, and four results produced from the previous pass that need to be fed back into the filter. Result four is the actual output of one pass through the loop, even though it is used to affect the output of the next pass. Normally, the four results would be kept internally in a software implementation, with only the fourth being returned, and would

not have to passed each time the filter is used. However, the FPGA cannot move data upwards within itself. Therefore, the four results need to be forwarded to the function prior to its being called again. With this arrangement the FPGA function can operate the filter at any frequency without having to be reloaded or depending on a cooperating function.

A hypothetical FPGA function for the Gammatone Filter is shown in Figure 35. Each filter stage is performed in one bank. The execution of each filter consists primarily of three 16-bit by 16-bit multiplies which give 32-bit results, two 32-bit additions, followed by a shift to the right of 15 bits. The multiplies are independent of each other and can therefore be performed in parallel. Garp can perform such multiplies in five clock cycles[26], so the same is assumed true for FPGADLX. Unfortunately, the adds cannot execute in parallel because they are dependent. Addition of two 16-bit integers on Garp takes two cycles. [12] Since 32-bit integers are involved here, it is estimated that FPGADLX will need four cycles for each addition. Again turning to Garp, the 15-bit shift is expected to take only one cycle. [26] Fitting all four filters into the FPGA requires some clever movement of data. For example, the three constants (ap1, ap2, and ap3) must be present in each bank, but can only enter the array in certain banks and must, therefore, be transmitted to each bank. Since the longest transmission is across three banks, it is estimated that it will take at least three clock cycles from the time the function begins execution before the multiplies can commence. Adding together all the clock cycles gives a minimal latency of 17 cycles. Because the function is extremely dense and

complex, there is bound to be some overhead involved in moving data and synchronizing operations. Therefore, an extra 25% of the estimated 17 cycles (rounding up to the nearest integer) is added to give a final latency of 22 cycles. Because there are data dependencies in from one loop to another, it is useless to create a pipelined function.

Test results show that the software version took 19,023 cycles while the reconfigurable version required only 11,314 cycles. This means that the reconfigurable program was 1.68 times faster. Remarkably, though, the software program had a dynamic instruction count 11.22 times greater than the reconfigurable program (46,752 versus 4,165). The speedup and the difference in the instruction count indicate that the FPGA function is somewhat inefficient. It is believed that the inefficiencies occur mainly because of the latencies the FPGA additions and multiplications compared to their FU counterparts. Unfortunately, the multiplications and additions must be performed. At least executing all four filters at the same time partially hides the cost of the multiplies and adds. The overhead of distributing the constants across all banks of the array doesn't help the function's efficiency either. However, overhead of distributing the constants could be lessened by using a cooperating function, or functions, to pre-load the constants in advance of a running the filter at a certain frequency for a relatively long time.

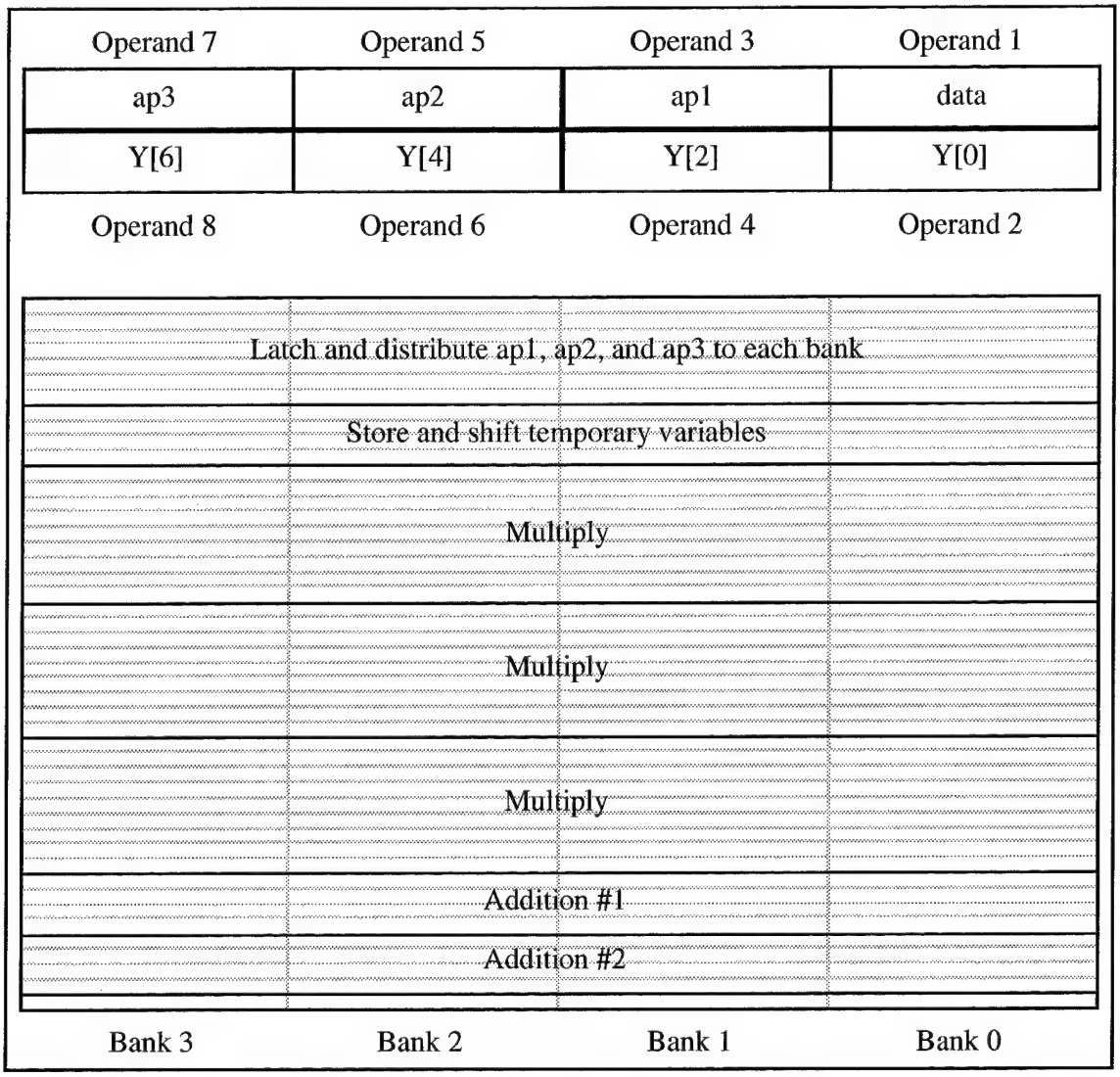

Figure 35. Gammatone Filter Function Layout. Shown above is the arrangement of data in an entry of the FPGA Issue Window and a hypothetical layout of the Gammatone Filter function in the FPGA. The function is four banks wide by 47 rows deep and has an estimated latency of 22 clock cycles.

6.4.3 Program Three: 2D Discrete Cosine Transform

The Two-Dimensional Discrete Cosine Transform (2D-DCT) is a common routine and forms a major part of the JPEG and MPEG image storage algorithms. In JPEG, specifically, the 2D-FFT is always applied to an image in 8x8 pixel chunks and produces an 8x8 result. This test program implements a 2D-DCT on an 8x8 array as

might be found in a JPEG program. The program works by performing a 1D-DCT on all eight rows of the input array, transposing the intermediate results, and then doing another 1D-DCT on the columns (which are now in rows because of the transpose). The 1D-DCT used in this program is from an algorithm proposed in [36]. The source code for the 1D-DCT algorithm, optimized to eliminate redundant variables, is shown in Figure 36. A nice thing about this 1D-DCT algorithm is that it requires only the eight integers as input—all of which can be given to an FPGA function at once.

```
b[0] = a[0] + a[7];
b[1] = a[1] + a[6];
b[2] = a[2] - a[4];
b[3] = a[1] - a[6];
b[4] = a[2] + a[5];
b[5] = a[3] + a[4];
b[6] = a[2] - a[5];
b[7] = a[0] - a[7];

c[0] = b[0] + b[5];
c[1] = b[1] - b[4];
c[2] = b[2] + b[6];
c[3] = b[1] + b[4];
c[4] = b[0] - b[5];
c[5] = b[3] + b[7];
c[6] = b[3] + b[6];

d[0] = c[0] + c[3];
d[1] = c[0] - c[3];
d[3] = c[1] + c[4];
d[4] = c[2] - c[5];

e[2] = M3 * c[2];
e[3] = M1 * c[6];
e[4] = M4 * c[5];
e[6] = M1 * d[3];
e[7] = M2 * d[4];

f[2] = c[4] + e[6];
f[3] = c[4] - e[6];
f[4] = b[7] + e[3];
f[5] = b[7] - e[3];
f[6] = e[2] + e[7];
f[7] = e[4] + e[7];

s[0] = d[0];
s[1] = f[4] + f[7];
s[2] = f[2];
s[3] = f[5] - f[6];
s[4] = d[1];
s[5] = f[5] + f[6];
s[6] = f[3];
s[7] = f[4] - f[7];
```

Figure 36. 1D-DCT Source Code. M1 through M4 are 8-bit constants.

The reconfigurable version of the program uses an FPGA function to perform the 1D-DCTs. A simplified, hypothetical layout of the function is shown in Figure 37. The function is shown broken up into the computational stages of the algorithm. It is estimated that each 8-bit add takes one cycle. Since Garp can multiply an 8-bit constant by a 32-bit integer in two cycles [26], it is assumed that FPGADLX can do the DCT multiplies (an 8-bit constant

with an 8-bit integer) in two cycles. With these latencies for the adds and multiplies and taking into account the time to move results across banks, it is estimated that FPGADLX's FPGA can perform a 1D-DCT in 9 cycles.

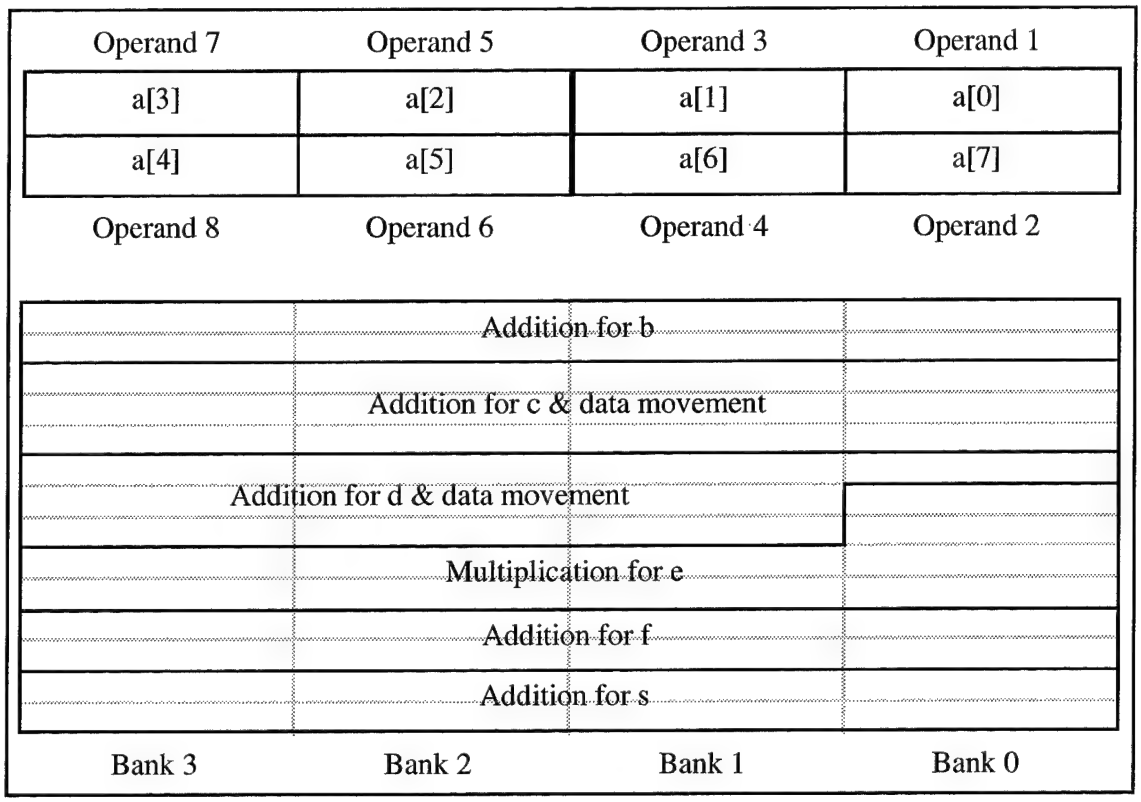

Figure 37. DCT Function Layout. At top is the positioning of operands in an entry of the FPGA Issue Window. The bottom picture shows a hypothetical layout of the 1D-DCT function in the FPGA. The function is four banks wide by 14 rows deep and has a nine clock cycle latency.

The alert reader will notice that the 1D-DCT algorithm produces eight 8-bit results (s[0] through s[7] in Figure 36) but that the FPGA can have at most four results. In order to get around this problem the FPGA function places two 8-bit results in each of its four outputs. The 8-bit results are recovered by shifting them out of function's results.

The performance item of interest in this test program is the time to perform the eight row and eight column 1D-DCTs—in other words, the 2D-DCT minus the transpose. The cycle count for 16 1D-DCTs in software was 1,169. The reconfigurable count was 515. Thus the reconfigurable version was 2.27 times faster than the software version.

6.4.4 Program Four: IDEA Encryption

The International Data Encryption Algorithm (IDEA) is the last test program for FPGADLX. IDEA has been suggested as a benchmark for measuring reconfigurable computer performance [12] which led to its selection for implementation on FPGADLX. IDEA encrypts and decrypts data in 64-bit blocks via the same computational methods, although encryption and decryption use different sub-keys generated from the same 128-bit master key. Each data block is split into four 16-bit sub-blocks and subjected to eight rounds of encryption. A round consists of XORing, adding modulo 2^{16}, and multiplying modulo $2^{16}+1$ the four sub-blocks with one another and with six 16-bit sub-blocks of the encryption or decryption key. Between each round the second and third sub-blocks are swapped. After all eight rounds are completed, the four sub-blocks are put through a final output transformation—a subset of the operations performed in a normal round—which uses the remaining four sub-blocks of the encryption/decryption key. Figure 38 shows the IDEA algorithm graphically.

The source code used for the test program is based on code presented in [47: 519-527]. The test program encodes and then decodes data one block at a time. This gives performance results similar to what one might expect by running IDEA in ciphertext feedback mode (CFM) where each block of data is XORed with the encrypted previous block before being encrypted or after being decrypted. Operating in CFM creates dependencies bewteen blocks which means that only one block can be encrypted or decrypted at one time. Thus, the test program gives a good idea of FPGADLX's performance at encrypting and decrypting individual blocks.

Another form of IDEA known as electronic code book mode (ECB) is free of dependencies between blocks. ECB allows multiple blocks of data to be encrypted or decrypted in a pipelined manner when specialized hardware is present, as it is in FPGADLX. Thus, IDEA in ECB mode would most certainly work faster than CFM. Unfortunately, attempts to create a software assembly of a pipelined ECB version of IDEA failed—it is suspected that the DLX compiler is to blame—and tests were only run in CFM.

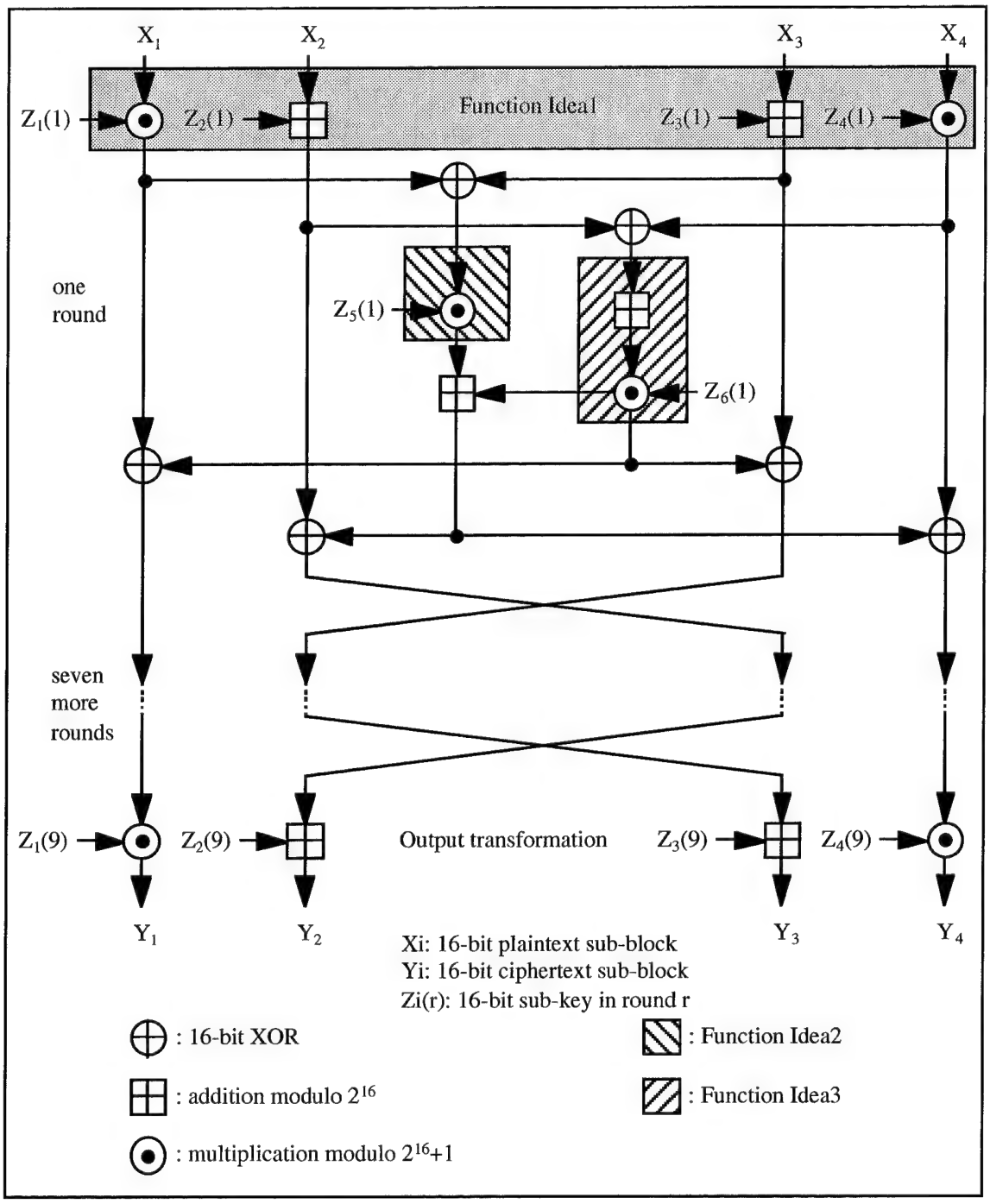

Figure 38. IDEA Encryption Algorithm. The algorithm encrypts a 64-bit block of data in eight rounds plus a final output transformation. The reconfigruable version of the algorithm uses three FPGA functions Idea1, Idea2, and Idea3 to perform portions of the encryption process as indicated in the diagram. Picture adapted from [47].

The test program measures the number of clock cycles needed for a function, *cipher_idea*, in the IDEA program to encrypt and then decrypt 10KB of data one 64-bit

block at a time (20KB of data altogether). The reconfigurable version of the test program implements some parts of *cipher_idea* in the FPGA and others in the the FUs already present in the processor. For the most part, the FPGA peforms all of the multiplications modulo $2^{16}+1$ and any adds that overlap with them and leaves all other operations to the fixed FUs. The modulo multiplications are ideal candidates for placement in the FPGA since their software equivalents require many dependence-laden operations. The other operations, the XORs and adds modulo 2^{16}, can be performed efficiently in the processor when data is typed as short integers (16-bit integers).

The reconfigurable IDEA assembly uses three functions Idea1, Idea2, and Idea3 during each round of encryption. Idea1 is also used for the final output transformation. The mapping of the three FPGA functions to the IDEA algorithm is shown in Figure 38. The functions are not pipelined. Idea1 and Idea2 have a latency of seven clock cycles. Idea3 has a latency of nine. A hypothetical layout of the three functions in the FPGA are shown in Figure 39. The shape, size, and latencies for the three functions are based on similar functions developed for IDEA on Garp [12].

The software IDEA program needed 927,246 cycles to process the 20KB of data. In comparison, the reconfigurable version required only 350,738 clocks which amounts to a 2.64 times increase in performance.

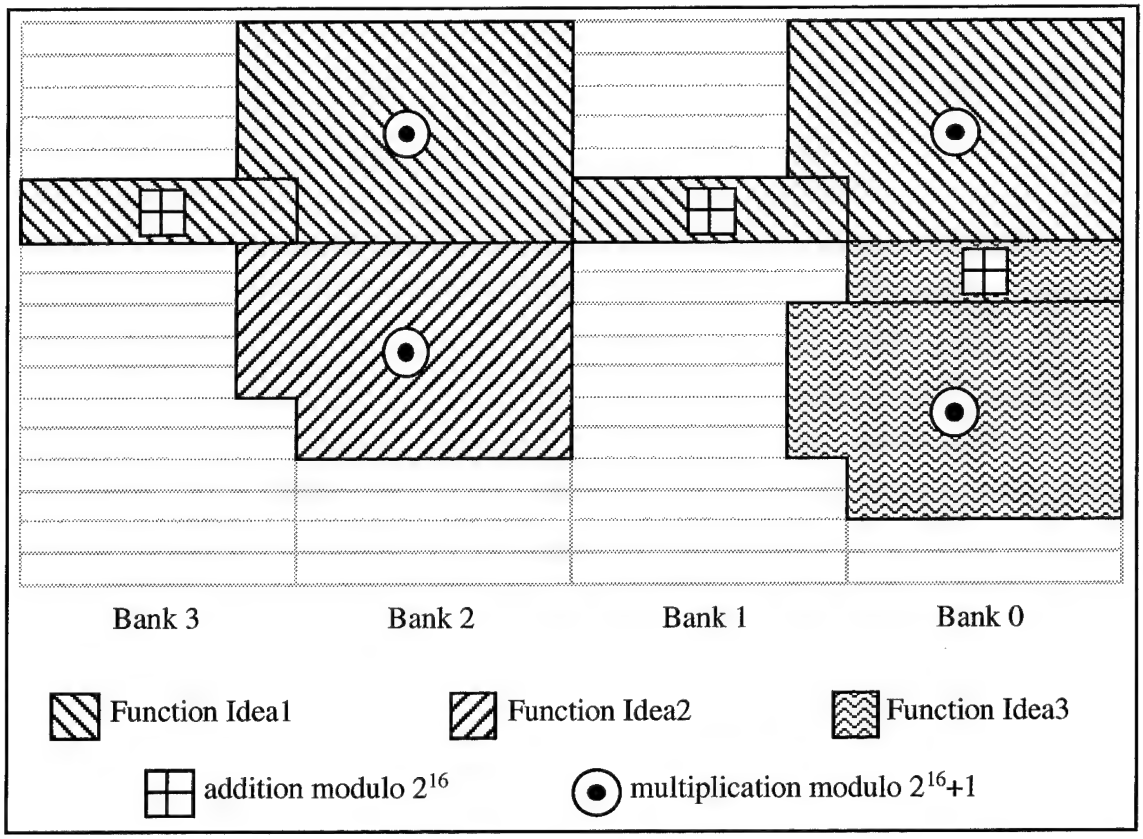

Figure 39. IDEA Functions in FPGA. The picture shows a hypothetical arrangement of the three functions for the IDEA encryption/decryption algorithm. The functions have latencies equal to their length in rows. The size, shape, and latencies of functions is based on work presented in [12]. Together the functions occupy a space that is 16 rows deep by four banks wide.

6.5 Analysis of Test Results

The FPGADLX design looks promising since each of the test programs showed some moderate amounts of speedup. Unfortunately, to try to gauge the overall effectiveness of FPGADLX from a handful of test programs would not be very scientific—the sample size is simply too small. Besides, the performance of each program is highly dependent on its implementation. For instance, better bit reversal routines exist. Still, the experience of constructing the four test programs, the results of

the programs themselves, and the design of FPGADLX can provide some insight into the importance of the architecture.

- **Actual speedups may be greater** — As mentioned previously, FPGADLX and its underlying host processor, SuperDLX, are perhaps more aggressive architectures than currently exist. With this understanding, the software versions of the test programs may run faster on FPGADLX than they would on a real processor. Therefore, if reconfigurable hardware of FPGADLX were added to a real processor and the reconfigurable test programs run, it is likely that the real processors would see greater speedups than FPGADLX since their superscalar core is not as aggressive as SuperDLX.

- **Wide width of FPGA is responsible for most speedup** — With the exception of the bit reversal program, the wide width of the FPGA array was the source of most of the speedup. This is especially evident when looking the Gammatone Filter and 2D-DCT programs, both of which used all four banks to perform near identical operations in parallel. If the FPGA had been a half or a quarter of its 128-bit width, then the time taken for each program's function would have potentially doubled or quadrupled, respectively. This would most likely have eliminated any increase in speed. One might conclude that modern multi-issue processors need a wide FPGA if they are to see any benefit from reconfigurable

computing. As a counterpoint, less aggressive processors, single-issue machines for example, can achieve speedup with a narrower FPGA array.

- **Functions for even one application consume much of the FPGA** — With the exception of the bit reversal function, the area used by the test applications' functions is a significant portion of the FPGA. The three IDEA functions combine to occupy one-quarter of the FPGA, as does the one DCT function. The Gammatone Filter function takes up almost three-fourths of the FPGA. If reconfigurable programs are going to regularly use so much room in the FPGA, then it is unlikely that more than a handful of programs (at most three or four) can execute reconfigurably at any given time. This number is too small and suggests that a paged FPGA or a larger FPGA would be immediately useful.

- **Programs can be rewritten to take advantage of the FPGA and FPGA functions** — Experience shows that programs may need to be restructured to obtain optimal reconfigurable performance. While simply converting an existing program into a reconfigurable version may provide some speedup, additional speedup may be obtained by using novel FPGA functions and/or rewriting the program to open more of it up to the FPGA. Two examples come to mind. First, the Gammatone Filter program used was just one of several implementations of the same algorithm and was picked because it allowed four filter stages to execute in parallel. The second example is the IDEA test program. In this case, though

not tested, higher speedups might be obtained by pipelining several blocks of data through the three IDEA functions when running in ECB mode.

VII. Conclusions and Recommendations

7.1 Research Goals and Contributions

The main goal of this thesis was to develop a high-level design for a general-purpose reconfigurable computer. This design included not just the hardware and the hardware's operation but also considered the operating system and compiler. The resulting design, called FPGADLX, includes many new ideas not found in any other reconfigurable computer effort to date. While FPGADLX is mainly conceptual and is not based on a real computer, the concepts presented should be easily transferable to actual superscalar or VLIW processor designs. There is enough detail in the design to enable aspects of its operation to be simulated and compared to a normal processor. Major accomplishments and contributions of this effort include:

- The requirements for a general-purpose reconfigurable computer system were derived from a study of past reconfigurable computing efforts and the characteristics of FPGA devices. Other requirements for the design came from considering the practical aspects of computer design such as user-friendliness, cost, and aesthetics. The gathered requirements allowed for various system models to be developed and judged. Eventually, one model was chosen to become the seed for the final FPGADLX design.

- The FPGADLX design encompasses four aspects of computer system design: hardware architecture, OS support, compiler issues, and system operation. Addressing a wide range of design issues makes this thesis unique among reconfigurable computing efforts; other projects have focused on only one or two of the aforementioned issues.

- FPGADLX is the first project to explore the ramifications of adding reconfigurable arrays to superscalar processors. Other efforts have added FPGAs to single-issue, pipelined machines and shown some respectable results. Single issue processors are obsolete by today's standards and obtaining speedup through reconfiguration is easy. However, modern processors, employing dynamic reordering of instructions and VLIW techniques, are capable of issuing and retiring multiple instructions per clock cycle. Thus, they are more powerful than the previous generation of single issue processors. Moreover, an FPGA supporting functions of varying latencies fits in well with a superscalar processor's ability to resolve dependencies between instructions at run time. This thesis has proposed a way to fit reconfigurable components into a superscalar, multi-issue machine and still achieve speedup—for some test cases.

- In addition to the basic FPGADLX design presented, several enhancements were suggested. These extras were seen as unnecessary for the elementary operation of FPGADLX and were therefore not part of the basic design. If implemented, the

enhancements should add to the power of the FPGADLX, but only at the expense of complicating the design.

- A simulation of FPGADLX was developed from a superscalar DLX simulator. The FPGADLX simulator doesn't fully model the FPGADLX design, but it is adequate enough to run sample programs and obtain rough estimates of FPGADLX's performance. The simulator was used to compare the clock cycle counts of normal DLX programs to their reconfigurable equivalents. Results from test programs show that FPGADLX is capable of achieving small to moderate speedups.

- The four-bank, 128-bit wide design of the FPGA allows functions to exploit parallelism in the algorithms they are implementing. This observation led to the postulation that the FPGA's wide width may be necessary for achieving speedup for reconfigurable superscalar processors. Regardless, dividing FPGA into four banks is worthwhile since it allows reconfigurable programs to tailor functions to their specific needs and helps to interface the FGPA into the host processor's execution model.

- The concepts of netlists that contain multiple functions and of cooperating functions are believed to be a first for reconfigurable computing. Cooperative functions are especially useful since they help overcome operand limitations and can force explicit dependencies between functions. Creating dependencies is important in an superscalar machine where functions can execute out of order.

7.2 Research Recommendations

Even though this thesis has made progress in reconfigurable computing, there are several areas which are ripe for additional study. Most of these are direct extensions of the FPGADLX architecture, work begun in this thesis, and problems discovered during this effort. Others are problems facing reconfigurable computing in general.

- **FPGA Design for FPGADLX** — The FPGA for FPGADLX was described in only the most general of terms. The action and features of the FPGA have been specified, but there is no physical design and layout for it. Until an exact design is made, there is no way to judge or model the true performance of FPGADLX.

- **Netlist Loader** — The Netlist Loader is a big part of the FPGADLX design and needs to be built. As described in Chapter 4, the Loader must be tightly integrated into the processor design and work in close conjunction with the FPGA.

- **Operating System** — Several additions to the duties normally performed by an OS were proposed for FPAGDLX. These included support for context switching and handling of netlist loads and unloads. Since the OS is a key part of any computer system, more attention needs to be placed on construction of an OS, not only for FPGADLX, but for reconfigurable computing in general.

- **Compiler** — Even for normal computer systems, compilation is one area in which research and improvements are continual. For reconfigurable systems

compilation is doubly important since programs consist of both a software and hardware portion. Unfortunately, the double-sided nature of reconfigurable compilation also makes it much tougher. Problems to be overcome include: identification of code that could run favorably in an FPGA, partitioning a program into software and reconfigurable parts, automatic generation of netlists from software code, inserting FPGA commands and optimizing code for FPGA execution, and the use of netlist libraries.

- **Improvements to the Simulator** — While perchance not a master's thesis itself, the FPGADLX simulator needs to be upgraded so that it more accurately models a full computer system and contains all the features of FPGADLX. Example additions include a cache structure and memory hierarchy, a more realistic implementation of issue and write back (the current model is too liberal), a true CDB, load forwarding, support for double precision float arithmetic, netlist loading and unloading, multi-tasking, and support for asynchronous functions.

- **Cost and Performance Tradeoffs** — This thesis has mentioned that the area of a chip housing a reconfigurable processor would be greater than its non-reconfigurable counterpart. Just how large the difference would be depends on the design and amount of FPGA logic included in the reconfigurable die. Of course, die size has a direct relationship to the cost of the die. This thesis has refrained from making any estimates about the size of the basic FPGADLX design and has instead focused on performance aspects. However, knowing the area consumed is

important to the final success of the design; increased performance must be justified in terms of the expense incurred to achieve it. Therefore, one direction for future research is to conduct a detailed cost versus performance study for FPGADLX and other reconfigurable processors. Further, reconfigurable computing could be compared to other ideas for increasing processor performance in order to better ascertain which gives better results per dollar spent.

7.3 Conclusion

The creation of a viable general-purpose reconfigurable computer is still many years in the future. There are simply too many problems to be solved and options to explore than one lone thesis effort can tackle. It was therefore decided that this thesis should take into account the obstacles facing reconfigurable computing when working up the design that eventually became FPGADLX. Hopefully, this has resulted in FPGADLX being a reasonable first attempt at a superscalar RC. Successive studies should be able to expand and refine the FPGADLX model and its simulator. Some years from now, the concepts presented herein might find their way into a real system.

One subject that critics of reconfigurable computing like to bring up is that reconfigurable computing is a novelty and won't be of any use except for some very limited applications. In response to the critics and also to develop the requirements for a viable RC, it was determined that there is not enough evidence to allow a solid prediction about the future of reconfigurable processing. At worst, reconfigurable computing is just

another option for increased performance. At best, it might revolutionize computing. Much work needs to be done to test and evaluate the performance of FPGADLX, or similar designs, in order to gain a better understanding of the merits and possibilities of reconfigurable computers. This involves trying to expand the application base of RCs—inventing new ways in which to use them—so that they become more indispensable for everyday computing.

Overall, this effort has achieved what it set out to do: develop a model for a viable general-purpose reconfigurable computer based on modern microprocessor designs. This involved the study of FPGA technology and its strengths and weaknesses, learning from the successes and mistakes of past projects, and examining the need and application base for reconfigurable systems. The background work allowed for the development of requirements that would help ensure the design of a successful RC system. Those requirements were then used to propose a model, FPGADLX. The design is a good starting point for the refinement, study, and, perhaps, eventual implementation of the world's first reconfigurable superscalar processor.

Appendix A: Addendum to SuperDLX: A Generic Superscalar Processor

A.1 Introduction

The FPGADLX simulator is based on a superscalar DLX simulator developed by Cecile Moura as a Master's effort while a student a McGill University in Montreal, Canada. Moura's simulator, SuperDLX, was also based on a DLX simulator created to accompany Hennessy and Patterson's Computer Architecture: A Quantitative Approach series of textbooks.

A report entitled SuperDLX: A Generic Superscalar Simulator (included as part of the SuperDLX simulator [41]) details the operation of the superscalar processor and construction of the simulator and gives directions for running the simulator. This appendix revises and adds to this report. Creating the FPGADLX simulator meant several changes had to be made to SuperDLX. The primary change to SuperDLX was to the operation of the functional units and the issue and write back logic. The main additions was the FPGA and the FPGA issue window. This appendix seeks only to relay the major changes made to SuperDLX. It does not go into great depth, but merely corrects SuperDLX where necessary and describes the code added for the FPGA. This appendix is divided into two parts: the corrections to SuperDLX and the additions. The corrections section focuses on the modifications to SuperDLX itself while the additions section is concerned with making the simulator reconfigurable.

A.2 Changes to SuperDLX

The changes made to SuperDLX centered on two things. The first is combining the Integer Unit with the Floating Point Unit. Second is the operation of the functional units (FUs). There are several other ancillary issues that will also be covered.

A.2.1 Issue Windows

Having two separate halves of processor, one for integer operations and the other for floating point (FP) operations, was seen as an unnecessary complication to the simulator. While some actual processors do have more than one issue window and divide operations by FP, integer, and memory types, there is no strong consensus as to what is the "right" or "wrong" way to structure a processor in this regard. Therefore, the Integer Unit and Floating Point Unit have been combined in FPGADLX. The changes are detailed below.

- The Integer Reorder Buffer and FP Reorder Buffer have been merged into one single buffer, now called just the Reorder Buffer. Although, in some places in the source code the Integer Reorder Buffer name remains.
- The central windows (issue windows) have also been merged into one window known as the Instruction Window.

- Because there is only one reorder buffer and one issue window, synchronizing the two buffers and windows for traps and branches is no longer necessary. The synchronizing code has been removed.

- Decoding of instructions occurs normally, except that all instructions enter the same reorder buffer and instruction window.

- Operand forwarding has been altered so that there is separation between FP and integer operands—the Reorder Buffer only stores register numbers, special care is needed to differentiate between FP and integer entries. Thus, integer instructions get forwarded integer values and FP instructions get only FP values.

A.2.2 Functional Units

As originally programmed, the SuperDLX's FUs did not resemble those found in actual superscalar processors. For example, a real processor will have one FP multiplier that has, say, five pipelined stages. With this arrangement, multiplies take five clock cycles to complete and a new multiply can enter the pipeline each clock cycle. In SuperDLX, however, a multiply might still have a latency of five cycles, but instead of one pipelined multiplier, there are several non-pipelined ones. Besides that, SuperDLX had some FUs that are not found as stand-alone FUs in normal processors. An example of this is the shifter and comparitor units. Normally, shifts and compares are performed by the arithmetic logic unit (ALU). FPGADLX fixes both of these problems.

- The default FUs for the simulator have changed. There are now two ALUs (add, subtract, shift, compare, and logical operations), two address adders, one integer branch unit (calculates target addresses for integer branches), one FP adder (also does FP-integer conversions), an FP multiplier, one FP divider, and an FP branch unit.

- Along with number and type, the operation of the FUs have changed as well to better imitate real hardware. FUs still have latencies, but are established as pipelined or non-pipelined. An FU consists of a several stages—one stage for each clock cycle of latency. The first stage is known as the first element and accepts the Reorder Buffer Tag (RBT) of an instruction entering an FU. Unused stages are given a negative value—all RBTs are greater than or equal to zero. Pipelined FUs are available for new instructions whenever their first element is negative. For non-pipelined FUs, every stage must be checked to determine if it can start a new instruction. If all stages contain negative numbers, then a non-pipelined FU is not busy. The operation of actual FUs is simulated by shifting RBTs from the first element to the last stage, the tail element. For single cycle operations (latency equal to one) the first and last elements are the same entity. An FU is ready to write back when its last element contains an RBT. An FU is frozen as long as an RBT remains in the last element. Write back is performed by placing a negative number into an FUs last element.

- The execution phase of the processor has been almost totally rewritten. An instruction still waits in the Instruction Window until they all of its operands are valid. When that condition is met, an FU appropriate for the instruction is searched for. If an FU is available, either **Execute Integer** or **Execute Float** is called to compute the result, which is then placed in the instruction's reorder buffer entry. The FU receives the instruction's RBT and the Reorder Buffer gets the number of the FU handling its instruction.

- For authenticity, the cycle when an instruction writes back is no longer pre-calculated and used to govern the write back process. Instead, write back proceeds by scanning a list of active Reorder Buffer entries (instructions which are executing in an FU) from head to tail. As each entry is scanned, the RBT of last element of the FU performing the calculation is checked to see if it matches the Reorder Buffer entry being scanned. A match signals causes the FU to write back; the Reorder Buffer entry is set to valid and the last element of the FU is cleared.

- When recovering from a mispredicted branch, the FU stage corresponding to a Reorder Buffer entry being flushed is set to -1. FU stages for entries older than the mispredicted branch are unaffected.

- The use and operation of FUs is totally within the file *supersim.c*. Moving RBTs through the FUs is done by calling **IncrementPipelinedFUs**. Finding an FU of the right type for an instruction and determining whether or not an FU of needed kind is available is done by calling **SearchForOpenFU**. Loading and freeing FUs

of RBTs is done in **Execute** and **WriteBack**, respectively. FUs are created and initialized in **Sim_Create** and **Sim_Init**, both of which are in *sim.c*.

A.2.3 Miscellaneous Changes

- **WriteBack** has been modified so that the maximum number of write backs is the same as the maximum number of instructions allowed to enter execution each cycle. The default value is four, which is also the number of decodes and commits allowed each cycle.

- SuperDLX allowed certain default settings to be modified by reading in a configuration file once the simulator was running. The changes to the FUs broke this capability. *Read.c*, the file containing the code for this, has been patched to support FPGADLX and to make compilation. However, it has never been tested since the default settings were always used. It is doubtful that reading in a configuration file with the new *read.c* will work.

- The X windows interface has also been patched, but only to enable compilation. Running FPGADLX under X will probably not work.

- The printing of statistics and machine execution (status of the various queues, windows, and the Reorder Buffer) have been altered to reflect architectural changes made.

- The simulator accepts the same commands as specified in "Appendix A: User Manual" of SuperDLX. The only exceptions being commands for the X interface and reading the configuration file.

- Several data structures in *dlx.h* were modified either to add new variables, delete obsolete variables, or change the use of a variable. The meaning of each new or changed variable has been commented in the code.

A.3 Additions for Reconfiguration

Making SuperDLX into FPGADLX required the modification and/or addition of several files and functions to the simulator. In most cases the purpose of the additions should be clear from reading Chapter 4 of the thesis. Where things are not so obvious, they will be explained. Because the changes were over such a large area it is best to address them file-by-file and function-by-function.

- ***asm.c*** — An entry for "fpga" has been added to the opcodes table. Because of the FPGA execute instruction, instructions have been changed to have four operands in **Asm_Assemble**. This has not impact on normal DLX instructions. The ability to assemble and dis-assemble FPGA execute instructions has been added to **Asm_Assemble** and **Asm_Disassemble**, respectively.

- ***fpga.h*** — This file contains macros for FPGA functions. Each function—16 at the present time although an unlimited amount may exist—is given a name and associated with a number.

- ***fpgacalc.c*** — This file is much like *supercalc.c* in many respects. The file contains a table *fpgaFunctions* which includes all the information needed to create an FPGA function. **RemoveFPGAOperation** removes FPGA functions from the FPGA instruction window. **ExecuteFPGA** is modeled after **ExecuteInteger** and **ExecuteFloat** and performs the work done by an FPGA function. Each function in ExecuteFPGA is an item in a switch on the variable absFuncNum (absolute function number). absFuncNum allows for there to be more the 1024 functions in the switch structure by enabling function numbers from programs to be renamed based on some table (looking ahead to simulating large programs and supporting multi-tasking). Presently, there is no renaming although the structure for it exists. Functions make use of the eight operands provided—not all operands have to be used. The functions are responsible for placing results in the correct Reorder Buffer entries. **ExecuteFPGA** is called by the **Execute** routine found in **supersim.c**.

- ***sim.c*** — **Sim_Init** initializes each of the FPGA functional units (FFUs). In FPGADLX each FPGA function is treated as if it were four functional units, although the four FUs are grouped together in one functional unit, the FFU. An FFU behaves in the same manner as a normal FU except that all four FUs in the FFU must meet test conditions. For example, all four entries in the last element of an FFU must be negative for the pipeline to flow. Just one non-negative entry will stall an FPGA function's pipeline. The **Sim_Create** routine builds the

FPGA Issue Window and the stages of each FFU through calls to **CreateFPGA_FUStages**. In addition to its other duties, **Reset** frees the memory used by the FFUs. **Compile** has been modified to take an assembled FPGA execute instruction (placed in the simulator's memory by **asm.c**) and restructure it for use by the simulator. This was already done for other DLX instructions, but FPGA instructions required some changes. An entry for the FPGA instruction has also been added to the opTable structure.

- ***superdecode.c*** — Decoding of FPGA execute instructions, and thus sequences, is handled by code added to **DecodeInstruction**. Instructions are decoded—entered into the FPGA Instruction Window and the Reorder Buffer—one at a time. The decoder performs all the activities needed to decode instructions in a sequence into the same entry of entry in the Instruction Window. This includes checking for errors in function numbers and in signaling the end of a sequence, opening a new entry when beginning a new sequence, closing an entry when a sequence is completed, and forwarding operands. Forwarding operands is done by calling **FPGASearchOperand** which repackages FPGA Instruction Window operands for a call to **SearchOperand** (nice code resuse!). Entering FPGA instructions in the Reorder Buffer is done by a call to **EnterRBuffer** just like normal instructions.

- ***supercalc.c*** — The **BranchExecute** function was modified to flush incompletely decoded sequences from the FPGA Instruction Window and to flush FFUs. This

is part of the corrective action taken by the processor to recover from bad branch predictions.

- ***supersim.c*** — As might be expected, there were many changes to this file. The changes were mainly confined to the **Execute** and **WriteBack** functions, although the infOp and intUnit tables were updated for the FPGA.

  Additions to **Execute** include removing flushed FPGA Instruction Window entries, forwarding operands to the FPGA Window, scanning the FPGA Window for ready entries, and issuing (begin execution) ready entries to the FPGA. These are, in essence, the same steps carried out for normal instructions; execution of FPGA functions is procedurally different only in that there are more operands to check and Reorder Buffer entries to manipulate. One new routine called by **Execute** is **SearchForOpenFPGA_FU**. This function checks the status of an FFU to see if it is ready to accept new data. When it has been determined that an FPGA function is ready to execute, **ExecuteFPGA** (in *fpgacalc.c*) is called.

  To **WriteBack**, code was added to detect that a Reorder Buffer entry's result is ready in an FFU. If one of the four RBTs in the last element of the FFU matches the number of the Reorder Buffer entry being examined, then the Reorder Buffer entry is marked as valid. If the entry being written back was the last in an FPGA sequence, then all four RBTs in the last element of the FFU are reset to -1. At the end of **WriteBack** a call is made to **IncrementFPGAUnits**. This function serves the same purpose as **IncrementPipelinedFUs** but for FFUs.

- ***superdlx.h*** — *Superdlx.h* contains many of the macros and table structure definitions used by the simulator. Several macros and a structure for FPGA function information were added to this file.

- ***dlx.h*** — *Dlx.h* holds the data structures for the simulator. To this file was added definitions for an FPGA pipeline stage, an FFU, FPGA Instruction Window elements, and the FPGA Instruction Window. The DLX structure has been modified to contain the FFUs and the FPGA Instruction Window. Several macro definitions have also been added to *dlx.h*. They are the maximum number of FPGA functions allowed (at present 16, but the number is unlimited) and the opcode number for FPGA execute instructions (110). The process identification (PID) register has also been added by increasing the number of special purpose registers and assigning the macro definition PID_REG. The PID register is not used by the simulator at the present time and has been included in anticipation of future enhancements to the simulator.

Appendix B: Test Code

B.1 Introduction

This appendix contains the source code, software assembly, and reconfigurable assembly for the test programs covered in Chapter 6. To save space and reduce clutter, the assembly file the headers and footers have been removed. The deleted material consists of information needed by the simulator and it is not critical to understanding any of the programs.

B.2 Program One: Bit Reversal of 10-bit Integers

B.2.1 Source Code

```
#define SIZE 1024

main()
{
  unsigned int array[SIZE], i, j, input, output;

for (j=0; j<SIZE; ++j)
  {
    input = j;
    output = 0;

    for (i=0; i<10; i++)
    {
      output <<= 1;
      output = (input & 0x1) | output;
      input >>= 1;
    }

    array[j] = output;
  }

}  /* end main() */
```

B.2.2 Assembly

```
_main:
      ;; Initialize Stack Pointer
      add r14,r0,r0
      lhi r14, ((memSize-4)>>16)&0xffff
      addui r14, r14, ((memSize-4)&0xffff)
      ;; Save the old frame pointer
      sw -4(r14),r30
      ;; Save the return address
      sw -8(r14),r31
      ;; Establish new frame pointer
      add r30,r0,r14
      ;; Adjust Stack Pointer
      addi r14,r14,#-4136
      ;; Save Registers
      sw 0(r14),r3
      sw 4(r14),r4
      sw 8(r14),r5
      sw 12(r14),r6
      sw 16(r14),r7
      sw 20(r14),r8
      sw 24(r14),r9
      sw 28(r14),r10
      addi r8,r0,#0
      addi r9,r0,#1023
      sgtu  r1,r8,r9
      bnez r1,L11
      nop
      addi r10,r0,#-4104
      addi r9,r0,#1023
L9:
      add r7,r0,r8
      addi r4,r0,#0
      add r5,r0,r4
      addi r6,r0,#9
      sgtu  r1,r5,r6
      bnez r1,L10
      nop
      addi r6,r0,#9
L8:
      slli r4,r4,#1
      and r3,r7,#1
      or r4,r3,r4
      srli r7,r7,#1
      add r5,r5,#1
      sleu  r1,r5,r6
      bnez r1,L8
      nop
L10:
      slli r3,r8,#2
      add r3,r30,r3
```

```
        add r3,r3,r10
        sw 0(r3),r4
        add r8,r8,#1
        sleu  r1,r8,r9
        bnez r1,L9
        nop
L11:
        ;; Restore the saved registers
        lw r3,-4136(r30)
        nop
        lw r4,-4132(r30)
        nop
        lw r5,-4128(r30)
        nop
        lw r6,-4124(r30)
        nop
        lw r7,-4120(r30)
        nop
        lw r8,-4116(r30)
        nop
        lw r9,-4112(r30)
        nop
        lw r10,-4108(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; HALT
        jal _exit
        nop

_exit:
        trap #0
        jr r31
        nop
```

B.2.3 Reconfigurable Assembly

```
_main:
        ;; Initialize Stack Pointer
        add r14,r0,r0
        lhi r14, ((memSize-4)>>16)&0xffff
        addui r14, r14, ((memSize-4)&0xffff)
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
```

```
        sw -8(r14),r31
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-4136
        ;; Save Registers
        sw 0(r14),r3
        sw 4(r14),r4
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        addi r8,r0,#0
        addi r9,r0,#1023
        sgtu  r1,r8,r9
        bnez r1,L11
        nop
        addi r10,r0,#-4104
        addi r9,r0,#1023
L9:
        add r7,r0,r8
        addi r4,r0,#0
        add r5,r0,r4
        addi r6,r0,#9
        sgtu  r1,r5,r6
        bnez r1,L10
        nop
        addi r6,r0,#9
L8:
        fpga r4,r7,r0,#1031
L10:
        slli r3,r8,#2
        add r3,r30,r3
        add r3,r3,r10
        sw 0(r3),r4
        add r8,r8,#1
        sleu  r1,r8,r9
        bnez r1,L9
        nop
L11:
        ;; Restore the saved registers
        lw r3,-4136(r30)
        nop
        lw r4,-4132(r30)
        nop
        lw r5,-4128(r30)
        nop
        lw r6,-4124(r30)
        nop
        lw r7,-4120(r30)
        nop
        lw r8,-4116(r30)
```

```
        nop
        lw r9,-4112(r30)
        nop
        lw r10,-4108(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; HALT
        jal _exit
        nop

_exit:
        trap #0
        jr r31
        nop
```

B.3 Program Two: Gammatone Filter

B.3.1 Source Code

```
/* Constants for a 1Khz filter */
#define AP1 ((float) 0.024755)
#define AP2 ((float) 1.824484)
#define AP3 ((float) -0.920042)

#define TWO_TO_16TH_POWER 65536
#define TWO_TO_15TH_POWER 32768

#define ap1 ((int) (AP1 * TWO_TO_15TH_POWER))
#define ap2 ((int) (AP2 * TWO_TO_15TH_POWER))
#define ap3 ((int) (AP3 * TWO_TO_15TH_POWER))

void main ()
{
  int n;
  int X[4], Y[8];
  int output_store;
  int data;

  X[0] = X[1] = X[2] = X[3] = 0;
  Y[0] = Y[1] = Y[2] = Y[3] = Y[4] = Y[5] = Y[6] = Y[7] = 0;

  data = 30000;
  output_store =((X[0] * ap1) + (ap2 * Y[0]) + (ap3 * Y[1]))>> 15;
```

```
    X[0] = data;
    Y[1] = Y[0];
    Y[0] = output_store;

    output_store =((X[1] * ap1) + (ap2 * Y[2]) + (ap3 * Y[3]))>> 15;
    X[1] = Y[0];
    Y[3] = Y[2];
    Y[2] = output_store;

    output_store =((X[2] * ap1) + (ap2 * Y[4]) + (ap3 * Y[5]))>> 15;
    X[2] = Y[2];
    Y[5] = Y[4];
    Y[4] = output_store;

    output_store =((X[3] * ap1) + (ap2 * Y[6]) + (ap3 * Y[7]))>> 15;
    X[3] = Y[4];
    Y[7] = Y[6];
    Y[6] = output_store;

    for (n=0; n<512; n++)
    {

      data = 0;
      output_store =((X[0] * ap1) + (ap2 * Y[0]) + (ap3 * Y[1]))>> 15;
      X[0] = data;
      Y[1] = Y[0];
      Y[0] = output_store;

      output_store =((X[1] * ap1) + (ap2 * Y[2]) + (ap3 * Y[3]))>> 15;
      X[1] = Y[0];
      Y[3] = Y[2];
      Y[2] = output_store;

      output_store =((X[2] * ap1) + (ap2 * Y[4]) + (ap3 * Y[5]))>> 15;
      X[2] = Y[2];
      Y[5] = Y[4];
      Y[4] = output_store;

      output_store =((X[3] * ap1) + (ap2 * Y[6]) + (ap3 * Y[7]))>> 15;
      X[3] = Y[4];
      Y[7] = Y[6];
      Y[6] = output_store;

    }
}  /* end main() */
```

B.3.2 Assembly

```
_main:
      ;; Initialize Stack Pointer
      add r14,r0,r0
```

```
lhi r14, ((memSize-4)>>16)&0xffff
addui r14, r14, ((memSize-4)&0xffff)
;; Save the old frame pointer
sw -4(r14),r30
;; Save the return address
sw -8(r14),r31
;; Establish new frame pointer
add r30,r0,r14
;; Adjust Stack Pointer
addi r14,r14,#-104
;; Save Registers
sw 0(r14),r3
sw 4(r14),r4
sw 8(r14),r5
sw 12(r14),r6
sw 16(r14),r7
sw 20(r14),r8
sw 24(r14),r9
sw 28(r14),r10
sw 32(r14),r11
sw 36(r14),r12
sw 40(r14),r13
add r3,r30,#-24
sw 12(r3),r0
sw 8(r3),r0
sw 4(r3),r0
sw -24(r30),r0
add r3,r30,#-56
sw 28(r3),r0
sw 24(r3),r0
sw 20(r3),r0
sw 16(r3),r0
sw 12(r3),r0
sw 8(r3),r0
sw 4(r3),r0
sw -56(r30),r0
lw r7,-24(r30)
addi r5,r0,#811
movi2fp f2,r7
movi2fp f3,r5
mult f2,f2,f3
movfp2i r7,f2
addi r3,r0,#0
lhi r4,(59784>>16)&0xffff
addui r4,r4,(59784&0xffff)
lw r8,-52(r30)
addi r6,r0,#-30147
movi2fp f2,r8
movi2fp f3,r6
mult f2,f2,f3
movfp2i r8,f2
add r7,r7,r8
srai r7,r7,#15
addi r13,r0,#30000
```

```
sw -24(r30),r13
sw -52(r30),r3
sw -56(r30),r7
lw r7,-20(r30)
movi2fp f2,r7
movi2fp f3,r5
mult f2,f2,f3
movfp2i r7,f2
lw r8,-48(r30)
movi2fp f2,r8
movi2fp f3,r4
mult f2,f2,f3
movfp2i r9,f2
add r7,r7,r9
lw r9,-44(r30)
movi2fp f2,r9
movi2fp f3,r6
mult f2,f2,f3
movfp2i r9,f2
add r7,r7,r9
srai r7,r7,#15
lw r13,-56(r30)
sw -20(r30),r13
sw -44(r30),r8
sw -48(r30),r7
lw r7,-16(r30)
movi2fp f2,r7
movi2fp f3,r5
mult f2,f2,f3
movfp2i r7,f2
lw r8,-40(r30)
movi2fp f2,r8
movi2fp f3,r4
mult f2,f2,f3
movfp2i r9,f2
add r7,r7,r9
lw r9,-36(r30)
movi2fp f2,r9
movi2fp f3,r6
mult f2,f2,f3
movfp2i r9,f2
add r7,r7,r9
srai r7,r7,#15
lw r13,-48(r30)
sw -16(r30),r13
sw -36(r30),r8
sw -40(r30),r7
lw r7,-12(r30)
movi2fp f2,r7
movi2fp f3,r5
mult f2,f2,f3
movfp2i r7,f2
lw r5,-32(r30)
movi2fp f2,r5
```

```
        movi2fp f3,r4
        mult f2,f2,f3
        movfp2i r4,f2
        add r7,r7,r4
        lw r4,-28(r30)
        movi2fp f2,r4
        movi2fp f3,r6
        mult f2,f2,f3
        movfp2i r4,f2
        add r7,r7,r4
        srai r7,r7,#15
        lw r13,-40(r30)
        sw -12(r30),r13
        sw -28(r30),r5
        sw -32(r30),r7
        add r8,r0,r3
        addi r3,r0,#511
        sgt   r1,r8,r3
        bnez r1,L6
        nop
        addi r12,r0,#0
        addi r11,r0,#811
        lhi r10,(59784>>16)&0xffff
        addui r10,r10,(59784&0xffff)
        addi r9,r0,#-30147
L5:
        lw r3,-24(r30)
        movi2fp f2,r3
        movi2fp f3,r11
        mult f2,f2,f3
        movfp2i r3,f2
        lw r5,-56(r30)
        movi2fp f2,r5
        movi2fp f3,r10
        mult f2,f2,f3
        movfp2i r4,f2
        add r3,r3,r4
        lw r4,-52(r30)
        movi2fp f2,r4
        movi2fp f3,r9
        mult f2,f2,f3
        movfp2i r4,f2
        add r3,r3,r4
        srai r7,r3,#15
        add r4,r0,r7
        sw -24(r30),r12
        sw -52(r30),r5
        sw -56(r30),r7
        lw r3,-20(r30)
        movi2fp f2,r3
        movi2fp f3,r11
        mult f2,f2,f3
        movfp2i r3,f2
        lw r6,-48(r30)
```

```
movi2fp f2,r6
movi2fp f3,r10
mult f2,f2,f3
movfp2i r5,f2
add r3,r3,r5
lw r5,-44(r30)
movi2fp f2,r5
movi2fp f3,r9
mult f2,f2,f3
movfp2i r5,f2
add r3,r3,r5
srai r7,r3,#15
add r5,r0,r7
sw -20(r30),r4
sw -44(r30),r6
sw -48(r30),r7
lw r3,-16(r30)
movi2fp f2,r3
movi2fp f3,r11
mult f2,f2,f3
movfp2i r3,f2
lw r6,-40(r30)
movi2fp f2,r6
movi2fp f3,r10
mult f2,f2,f3
movfp2i r4,f2
add r3,r3,r4
lw r4,-36(r30)
movi2fp f2,r4
movi2fp f3,r9
mult f2,f2,f3
movfp2i r4,f2
add r3,r3,r4
srai r7,r3,#15
add r4,r0,r7
sw -16(r30),r5
sw -36(r30),r6
sw -40(r30),r7
lw r3,-12(r30)
movi2fp f2,r3
movi2fp f3,r11
mult f2,f2,f3
movfp2i r3,f2
lw r5,-32(r30)
movi2fp f2,r5
movi2fp f3,r10
mult f2,f2,f3
movfp2i r6,f2
add r3,r3,r6
lw r6,-28(r30)
movi2fp f2,r6
movi2fp f3,r9
mult f2,f2,f3
movfp2i r6,f2
```

```
        add r3,r3,r6
        srai r7,r3,#15
        sw -12(r30),r4
        sw -28(r30),r5
        sw -32(r30),r7
        add r8,r8,#1
        addi r3,r0,#511
        sle   r1,r8,r3
        bnez r1,L5
        nop
L6:
        ;; Restore the saved registers
        lw r3,-104(r30)
        nop
        lw r4,-100(r30)
        nop
        lw r5,-96(r30)
        nop
        lw r6,-92(r30)
        nop
        lw r7,-88(r30)
        nop
        lw r8,-84(r30)
        nop
        lw r9,-80(r30)
        nop
        lw r10,-76(r30)
        nop
        lw r11,-72(r30)
        nop
        lw r12,-68(r30)
        nop
        lw r13,-64(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; HALT
        jal _exit
        nop

_exit:
        trap #0
        jr r31
        nop
```

B.3.3 Reconfigurable Assembly

```
_main:
      ;; Initialize Stack Pointer
      add r14,r0,r0
      lhi r14, ((memSize-4)>>16)&0xffff
      addui r14, r14, ((memSize-4)&0xffff)
      ;; Save the old frame pointer
      sw -4(r14),r30
      ;; Save the return address
      sw -8(r14),r31
      ;; Establish new frame pointer
      add r30,r0,r14
      ;; Adjust Stack Pointer
      addi r14,r14,#-104
      ;; Save Registers
      sw 0(r14),r3
      sw 4(r14),r4
      sw 8(r14),r5
      sw 12(r14),r6
      sw 16(r14),r7
      sw 20(r14),r8
      sw 24(r14),r9
      sw 28(r14),r10
      sw 32(r14),r11
      sw 36(r14),r12
      sw 40(r14),r13
      ;; array X not needed
      ;; add r3,r30,#-24
      ;; sw 12(r3),r0
      ;; sw 8(r3),r0
      ;; sw 4(r3),r0
      ;; sw -24(r30),r0
      add r3,r30,#-56
      ;; sw 28(r3),r0 -- Y[7] not needed
      sw 24(r3),r0
      ;; sw 20(r3),r0 -- Y[5] not needed
      sw 16(r3),r0
      ;; sw 12(r3),r0 -- Y[3] not needed
      sw 8(r3),r0
      ;; sw 4(r3),r0 -- Y[1] not needed
      sw -56(r30),r0
      addi r3,r0,#0
      ;; load ap1
      addi r11,r0,#811
      ;; load ap2
      lhi r10,(59784>>16)&0xffff
      addui r10,r10,(59784&0xffff)
      ;; load ap3
      addi r9,r0,#-30147
      ;; load data
      addi r12,r0,#30000
```

```
        ;;
        ;; Start FPGA code
        ;;
        ;; load Y[0]
        lw r5,-56(r30)
        ;; load Y[2]
        lw r6,-48(r30)
        ;; load Y[4]
        lw r4,-40(r30)
        ;; load Y[6]
        lw r7,-32(r30)
        ;; call FPGA function APGXX
        fpga r5,r5,r12,#8
        fpga r6,r6,r11,#8
        fpga r4,r4,r10,#8
        fpga r7,r7,r9,#1032
        ;; store Y[0]
        sw -56(r30),r5
        ;; store Y[2]
        sw -48(r30),r6
        ;; store Y[4]
        sw -40(r30),r4
        ;; store Y[6]
        sw -32(r30),r7
        ;;
        ;; end first FPGA function
        ;;
        addi r3,r0,#511
        sgt   r1,r8,r3
        bnez r1,L6
        nop
        addi r12,r0,#0
L5:
        ;; call FPGA function APGXX
        fpga r5,r5,r12,#8
        fpga r6,r6,r11,#8
        fpga r4,r4,r10,#8
        fpga r7,r7,r9,#1032
        ;;
        ;; end loop body FPGA function
        ;;
        add r8,r8,#1
        addi r3,r0,#511
        sle   r1,r8,r3
        bnez r1,L5
        nop
        ;; save Y[0], Y[2], Y[4], Y[6]
        sw -56(r30),r5
        sw -48(r30),r6
        sw -40(r30),r4
        sw -32(r30),r7
L6:
        ;; Restore the saved registers
        lw r3,-104(r30)
```

```
        nop
        lw r4,-100(r30)
        nop
        lw r5,-96(r30)
        nop
        lw r6,-92(r30)
        nop
        lw r7,-88(r30)
        nop
        lw r8,-84(r30)
        nop
        lw r9,-80(r30)
        nop
        lw r10,-76(r30)
        nop
        lw r11,-72(r30)
        nop
        lw r12,-68(r30)
        nop
        lw r13,-64(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; HALT
        jal _exit
        nop

_exit:
        trap #0
        jr r31
        nop
```

B.4 Program Three: Discrete Cosine Transform

B.4.1 Source Code

```
void dct(char *array)
{
  /* variables */
  int i, j;
  char b[8], c[8], d[9], e[9], f[8];

  for(i=0; i<8; i++)
  {
```

```
    j = i*8;

    b[0] = array[j] + array[j+7];
    b[1] = array[j+1] + array[j+6];
    b[2] = array[j+2] - array[j+4];
    b[3] = array[j+1] - array[j+6];
    b[4] = array[j+2] + array[j+5];
    b[5] = array[j+3] + array[j+4];
    b[6] = array[j+2] - array[j+5];
    b[7] = array[j+0] - array[j+7];

    c[0] = b[0] + b[5];
    c[1] = b[1] - b[4];
    c[2] = b[2] + b[6];
    c[3] = b[1] + b[4];
    c[4] = b[0] - b[5];
    c[5] = b[3] + b[7];
    c[6] = b[3] + b[6];

    d[0] = c[0] + c[3];
    d[1] = c[0] - c[3];
    d[3] = c[1] + c[4];
    d[4] = c[2] - c[5];

    e[2] = M3 * c[2];
    e[3] = M1 * c[6];
    e[4] = M4 * c[5];
    e[6] = M1 * d[3];
    e[7] = M2 * d[4];

    f[2] = c[4] + e[6];
    f[3] = c[4] - e[6];
    f[4] = b[7] + e[3];
    f[5] = b[7] - e[3];
    f[6] = e[2] + e[7];
    f[7] = e[4] + e[7];

    array[j+0] = d[0];
    array[j+1] = f[4] + f[7];
    array[j+2] = f[2];
    array[j+3] = f[5] - f[6];
    array[j+4] = d[1];
    array[j+5] = f[5] + f[6];
    array[j+6] = f[3];
    array[j+7] = f[4] - f[7];
  }
}  /* dct() */


main()
{
  /* variables */
  char array[8][8];
  char temp;
```

```
  int i, j;

  /* Load data into array */
  for(i=0; i<8; i++)
    for(j=0; j<8; j++)
      array[i][j] = i+j;

#ifdef PRINT_OUT
  printf("\nInput array:\n");
  for(i=0; i<8; i++)
  {
    for(j=0; j<8; j++)
      printf("%4d ", array[i][j]);
    printf("\n");
  }
#endif

  /* Do row 1D-DCTs */
#ifdef COMPUTE
  dct(&array[0][0]);
#endif

#ifdef PRINT_OUT
  printf("\nArray after row DCT:\n");
  for(i=0; i<8; i++)
  {
    for(j=0; j<8; j++)
      printf("%4d ", array[i][j]);
    printf("\n");
  }
#endif

  /* Now transpose the array */
  for(i=0; i<8; i++)
    for(j=i; j<8; j++)
    {
      temp = array[i][j];
      array[i][j] = array[j][i];
      array[j][i] = temp;
    }

#ifdef PRINT_OUT
  printf("\nArray after 1st transpose:\n");
  for(i=0; i<8; i++)
  {
    for(j=0; j<8; j++)
      printf("%4d ", array[i][j]);
    printf("\n");
  }
#endif

  /* Do column DCTs */
#ifdef COMPUTE
  dct(&array[0][0]);
```

```
#endif

#ifdef PRINT_OUT
  printf("\nArray after column DCTs:\n");
  for(i=0; i<8; i++)
  {
    for(j=0; j<8; j++)
      printf("%4d ", array[i][j]);
    printf("\n");
  }
#endif

  /* Now transpose the array */
  for(i=0; i<8; i++)
    for(j=i; j<8; j++)
    {
      temp = array[i][j];
      array[i][j] = array[j][i];
      array[j][i] = temp;
    }

#ifdef PRINT_OUT
  printf("\nArray after 2nd transpose:\n");
  for(i=0; i<8; i++)
  {
    for(j=0; j<8; j++)
      printf("%4d ", array[i][j]);
    printf("\n");
  }
#endif

} /* main */
```

B.4.2 Assembly

```
LC0:
      .float 167.240059520000
      .align 4
.global _dct
_dct:
      ;; Save the old frame pointer
      sw -4(r14),r30
      ;; Save the return address
      sw -8(r14),r31
      ;; Establish new frame pointer
      add r30,r0,r14
      ;; Adjust Stack Pointer
      addi r14,r14,#-176
      ;; Save Registers
      sw 0(r14),r3
      sw 4(r14),r4
```

```
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        sw 32(r14),r11
        sw 36(r14),r12
        sw 40(r14),r13
        sw 44(r14),r15
        sw 48(r14),r16
        sw 52(r14),r17
        sw 56(r14),r18
        sw 60(r14),r19
        sw 64(r14),r20
        sw 68(r14),r21
        sw 72(r14),r22
        sw 76(r14),r23
        sw 80(r14),r24
        sw 84(r14),r25
        sw 88(r14),r26
        sw 92(r14),r27
        sw 96(r14),r28
        sw 100(r14),r29
        sf 104(r14),f4
        lw r25,0(r30)
        addi r22,r0,#0
        addi r3,r0,#7
        sgt   r1,r22,r3
        bnez r1,L6
        nop
        addi r24,r0,#7
        addi r28,r0,#2
        addi r29,r0,#4
        addi r27,r0,#5
        addi r23,r0,#90
L5:
        slli r3,r22,#3
        add r7,r25,r3
        add r3,r3,r25
        add r4,r3,r24
        lb r5,0(r7)
        lb r6,0(r4)
        add r5,r5,r6
        sb -16(r30),r5
        add r8,r3,#1
        add r6,r3,#6
        lb r13,0(r8)
        lb r9,0(r6)
        add r13,r13,r9
        sb -15(r30),r13
        add r10,r3,r28
        add r9,r3,r29
        lb r15,0(r10)
```

```
lb r11,0(r9)
sub r15,r15,r11
sb -14(r30),r15
lb r16,0(r8)
lb r11,0(r6)
sub r16,r16,r11
sb -13(r30),r16
add r11,r3,r27
lb r17,0(r10)
lb r12,0(r11)
add r17,r17,r12
sb -12(r30),r17
add r3,r3,#3
lb r19,0(r3)
lb r12,0(r9)
add r19,r19,r12
sb -11(r30),r19
lb r18,0(r10)
lb r12,0(r11)
sub r18,r18,r12
sb -10(r30),r18
lb r12,0(r7)
lb r20,0(r4)
sub r12,r12,r20
sb -9(r30),r12
add r20,r5,r19
sb -24(r30),r20
sub r21,r13,r17
sb -23(r30),r21
add r15,r15,r18
sb -22(r30),r15
add r13,r13,r17
sb -21(r30),r13
sub r5,r5,r19
sb -20(r30),r5
add r17,r16,r12
sb -19(r30),r17
add r16,r16,r18
sb -18(r30),r16
add r16,r20,r13
sb -40(r30),r16
sub r20,r20,r13
sb -39(r30),r20
add r21,r21,r5
sb -37(r30),r21
sub r15,r15,r17
sb -36(r30),r15
lb r13,-22(r30)
addi r15,r0,#69
movi2fp f2,r13
movi2fp f3,r15
mult f2,f2,f3
movfp2i r13,f2
sb -54(r30),r13
```

```
lb r15,-18(r30)
movi2fp f2,r15
movi2fp f3,r23
mult f2,f2,f3
movfp2i r15,f2
sb -53(r30),r15
lhi r1,(LC0>>16)&0xffff
addui r1,r1,(LC0&0xffff)
lf f4,0(r1)
cvtf2i f0,f4
movfp2i r18,f0
movi2fp f2,r18
movi2fp f3,r17
mult f2,f2,f3
movfp2i r18,f2
sb -52(r30),r18
lb r19,-37(r30)
movi2fp f2,r19
movi2fp f3,r23
mult f2,f2,f3
movfp2i r19,f2
sb -50(r30),r19
lb r20,-36(r30)
slli r17,r20,#1
add r17,r17,r20
slli r17,r17,#4
sb -49(r30),r17
add r20,r5,r19
sb -62(r30),r20
sub r5,r5,r19
sb -61(r30),r5
add r5,r12,r15
sb -60(r30),r5
sub r12,r12,r15
sb -59(r30),r12
add r13,r13,r17
sb -58(r30),r13
add r18,r18,r17
sb -57(r30),r18
sb 0(r7),r16
lb r5,-60(r30)
lb r7,-57(r30)
add r5,r5,r7
sb 0(r8),r5
lb r26,-62(r30)
sb 0(r10),r26
lb r5,-59(r30)
lb r7,-58(r30)
sub r5,r5,r7
sb 0(r3),r5
lb r26,-39(r30)
sb 0(r9),r26
lb r3,-59(r30)
lb r5,-58(r30)
```

```
        add r3,r3,r5
        sb 0(r11),r3
        lb r26,-61(r30)
        sb 0(r6),r26
        lb r3,-60(r30)
        lb r5,-57(r30)
        sub r3,r3,r5
        sb 0(r4),r3
        add r22,r22,#1
        sle   r1,r22,r24
        bnez r1,L5
        nop
L6:
        ;; Restore the saved registers
        lw r3,-176(r30)
        nop
        lw r4,-172(r30)
        nop
        lw r5,-168(r30)
        nop
        lw r6,-164(r30)
        nop
        lw r7,-160(r30)
        nop
        lw r8,-156(r30)
        nop
        lw r9,-152(r30)
        nop
        lw r10,-148(r30)
        nop
        lw r11,-144(r30)
        nop
        lw r12,-140(r30)
        nop
        lw r13,-136(r30)
        nop
        lw r15,-132(r30)
        nop
        lw r16,-128(r30)
        nop
        lw r17,-124(r30)
        nop
        lw r18,-120(r30)
        nop
        lw r19,-116(r30)
        nop
        lw r20,-112(r30)
        nop
        lw r21,-108(r30)
        nop
        lw r22,-104(r30)
        nop
        lw r23,-100(r30)
        nop
```

```
        lw r24,-96(r30)
        nop
        lw r25,-92(r30)
        nop
        lw r26,-88(r30)
        nop
        lw r27,-84(r30)
        nop
        lw r28,-80(r30)
        nop
        lw r29,-76(r30)
        nop
        lf f4,-72(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Return
        jr r31
        nop
LC1:
        .ascii "\12Input array:\12\0"
LC2:
        .ascii "%4d \0"
LC3:
        .ascii "\12\0"
LC4:
        .ascii "\12Array after row DCT:\12\0"
LC5:
        .ascii "\12Array after 1st transpose:\12\0"
LC6:
        .ascii "\12Array after column DCTs:\12\0"
LC7:
        .ascii "\12Array after 2nd transpose:\12\0"
        .align 4
.global _main
_main:
        ;; Initialize Stack Pointer
        add r14,r0,r0
        lhi r14, ((memSize-4)>>16)&0xffff
        addui r14, r14, ((memSize-4)&0xffff)
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r31
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-112
```

```
        ;; Save Registers
        sw 0(r14),r3
        sw 4(r14),r4
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        sw 32(r14),r11
        addi r7,r0,#0
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L87
        nop
        addi r10,r0,#-72
L15:
        addi r5,r0,#0
        addi r6,r0,#7
        sgt   r1,r5,r6
        bnez r1,L86
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r9,r3,r10
        add r8,r0,r7
        addi r6,r0,#7
L14:
        add r3,r9,r5
        add r4,r8,r5
        sb 0(r3),r4
        add r5,r5,#1
        sle   r1,r5,r6
        bnez r1,L14
        nop
L86:
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L15
        nop
L87:
        sub r14,r14,#8
        lhi r11,(LC1>>16)&0xffff
        addui r11,r11,(LC1&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        addi r7,r0,#0
        add r14,r14,#8
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L85
        nop
```

```
        addi r6,r0,#-72
L23:
        addi r5,r0,#0
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L84
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r4,r3,r6
L22:
        sub r14,r14,#8
        lhi r11,(LC2>>16)&0xffff
        addui r11,r11,(LC2&0xffff)
        sw 0(r14),r11
        add r3,r4,r5
        lb r3,0(r3)
        sw 4(r14),r3
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L22
        nop
L84:
        sub r14,r14,#8
        lhi r11,(LC3>>16)&0xffff
        addui r11,r11,(LC3&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L23
        nop
L85:
        sub r14,r14,#8
        add r3,r30,#-72
        sw 0(r14),r3
        jal _dct    ; 3
        nop
        lhi r11,(LC4>>16)&0xffff
        addui r11,r11,(LC4&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        addi r7,r0,#0
        add r14,r14,#8
        addi r3,r0,#7
        sgt   r1,r7,r3
```

```
        bnez r1,L83
        nop
        addi r6,r0,#-72
L31:
        addi r5,r0,#0
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L82
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r4,r3,r6
L30:
        sub r14,r14,#8
        lhi r11,(LC2>>16)&0xffff
        addui r11,r11,(LC2&0xffff)
        sw 0(r14),r11
        add r3,r4,r5
        lb r3,0(r3)
        sw 4(r14),r3
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L30
        nop
L82:
        sub r14,r14,#8
        lhi r11,(LC3>>16)&0xffff
        addui r11,r11,(LC3&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L31
        nop
L83:
        addi r7,r0,#0
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L81
        nop
        addi r9,r0,#-72
L39:
        add r5,r0,r7
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L80
        nop
```

```
        slli r3,r7,#3
        add r3,r30,r3
        add r8,r3,r9
L38:
        add r4,r8,r5
        lb r6,0(r4)
        slli r3,r5,#3
        add r3,r30,r3
        add r3,r3,r9
        add r3,r3,r7
        lb r11,0(r3)
        sb 0(r4),r11
        sb 0(r3),r6
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L38
        nop
L80:
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L39
        nop
L81:
        sub r14,r14,#8
        lhi r11,(LC5>>16)&0xffff
        addui r11,r11,(LC5&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        addi r7,r0,#0
        add r14,r14,#8
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L79
        nop
        addi r6,r0,#-72
L47:
        addi r5,r0,#0
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L78
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r4,r3,r6
L46:
        sub r14,r14,#8
        lhi r11,(LC2>>16)&0xffff
        addui r11,r11,(LC2&0xffff)
        sw 0(r14),r11
        add r3,r4,r5
        lb r3,0(r3)
```

```
        sw 4(r14),r3
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L46
        nop
L78:
        sub r14,r14,#8
        lhi r11,(LC3>>16)&0xffff
        addui r11,r11,(LC3&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L47
        nop
L79:
        sub r14,r14,#8
        add r3,r30,#-72
        sw 0(r14),r3
        jal _dct    ; 3
        nop
        lhi r11,(LC6>>16)&0xffff
        addui r11,r11,(LC6&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        addi r7,r0,#0
        add r14,r14,#8
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L77
        nop
        addi r6,r0,#-72
L55:
        addi r5,r0,#0
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L76
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r4,r3,r6
L54:
        sub r14,r14,#8
        lhi r11,(LC2>>16)&0xffff
        addui r11,r11,(LC2&0xffff)
        sw 0(r14),r11
```

```
        add r3,r4,r5
        lb r3,0(r3)
        sw 4(r14),r3
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L54
        nop
L76:
        sub r14,r14,#8
        lhi r11,(LC3>>16)&0xffff
        addui r11,r11,(LC3&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L55
        nop
L77:
        addi r7,r0,#0
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L75
        nop
        addi r9,r0,#-72
L63:
        add r5,r0,r7
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L74
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r8,r3,r9
L62:
        add r4,r8,r5
        lb r6,0(r4)
        slli r3,r5,#3
        add r3,r30,r3
        add r3,r3,r9
        add r3,r3,r7
        lb r11,0(r3)
        sb 0(r4),r11
        sb 0(r3),r6
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L62
```

```
        nop
L74:
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L63
        nop
L75:
        sub r14,r14,#8
        lhi r11,(LC7>>16)&0xffff
        addui r11,r11,(LC7&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        addi r7,r0,#0
        add r14,r14,#8
        addi r3,r0,#7
        sgt   r1,r7,r3
        bnez r1,L73
        nop
        addi r6,r0,#-72
L71:
        addi r5,r0,#0
        addi r3,r0,#7
        sgt   r1,r5,r3
        bnez r1,L72
        nop
        slli r3,r7,#3
        add r3,r30,r3
        add r4,r3,r6
L70:
        sub r14,r14,#8
        lhi r11,(LC2>>16)&0xffff
        addui r11,r11,(LC2&0xffff)
        sw 0(r14),r11
        add r3,r4,r5
        lb r3,0(r3)
        sw 4(r14),r3
        jal _printf ; 4
        nop
        add r14,r14,#8
        add r5,r5,#1
        addi r3,r0,#7
        sle   r1,r5,r3
        bnez r1,L70
        nop
L72:
        sub r14,r14,#8
        lhi r11,(LC3>>16)&0xffff
        addui r11,r11,(LC3&0xffff)
        sw 0(r14),r11
        jal _printf ; 4
        nop
        add r14,r14,#8
```

```
        add r7,r7,#1
        addi r3,r0,#7
        sle   r1,r7,r3
        bnez r1,L71
        nop
L73:
        ;; Restore the saved registers
        lw r3,-112(r30)
        nop
        lw r4,-108(r30)
        nop
        lw r5,-104(r30)
        nop
        lw r6,-100(r30)
        nop
        lw r7,-96(r30)
        nop
        lw r8,-92(r30)
        nop
        lw r9,-88(r30)
        nop
        lw r10,-84(r30)
        nop
        lw r11,-80(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; HALT
        jal _exit
        nop

_exit:
        trap #0
        jr r31
        nop
```

B.4.3 Reconfigurable Assembly

The reconfigurable assembly is the same as the software assembly from the previous section except for the section labeled `_dct:`. In the interest of space, only the `_dct:` portion of the reconfigurable assembly is reproduced here.

```
_dct:
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r31
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-176
        ;; Save Registers
        sw 0(r14),r3
        sw 4(r14),r4
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        sw 32(r14),r11
        sw 36(r14),r12
        sw 40(r14),r13
        sw 44(r14),r15
        sw 48(r14),r16
        sw 52(r14),r17
        sw 56(r14),r18
        sw 60(r14),r19
        sw 64(r14),r20
        sw 68(r14),r21
        sw 72(r14),r22
        sw 76(r14),r23
        sw 80(r14),r24
        sw 84(r14),r25
        sw 88(r14),r26
        sw 92(r14),r27
        sw 96(r14),r28
        sw 100(r14),r29
        sf 104(r14),f4
        lw r25,0(r30)
        addi r22,r0,#0
        addi r3,r0,#7
        sgt   r1,r22,r3
        bnez r1,L6
        nop
        addi r24,r0,#7
        addi r28,r0,#2
        addi r29,r0,#4
        addi r27,r0,#5
        addi r23,r0,#90
L5:
        slli r3,r22,#3
        ;; load data
        add r7,r25,r3
        add r3,r3,r25
        add r4,r3,r24
```

```
lb r12,0(r7)
lb r13,0(r4)
add r8,r3,#1
add r6,r3,#6
lb r14,0(r8)
lb r15,0(r6)
add r10,r3,r28
add r9,r3,r29
lb r16,0(r10)
lb r20,0(r9)
add r11,r3,r27
lb r18,0(r11)
add r3,r3,#3
lb r21,0(r3)
;; call FPGA DCT function
fpga r17,r12,r13,#9
fpga r19,r14,r15,#9
fpga r16,r16,r18,#9
fpga r18,r21,r20,#1033
;; store results
sb 0(r9),r16
srli r12,r16,#8
sb 0(r7),r12
sb 0(r4),r17
srli r13,r17,#8
sb 0(r8),r13
sb 0(r6),r18
srli r14,r18,#8
sb 0(r10),r14
sb 0(r3),r19
srli r15,r19,#8
sb 0(r11),r15
add r22,r22,#1
sle   r1,r22,r24
bnez r1,L5
nop
```

B.5 Program Four: IDEA Encryption Algorithm

B.5.1 Source Code

The source code for this program is found in two files: *idea.h* and *idea.c*. The two files are adapted from [47:519-527].

```
/* idea.h */
```

```
#include <stdio.h>
#include <string.h>

#define TEST
#define KBYTES 10
#define IDEAKEYSIZE 16
#define IDEABLOCKSIZE 8
#define word16 unsigned short int
#define word32 unsigned long int
#define boolean int
#define byte short int
#define ROUNDS 8
#define KEYLEN (6*ROUNDS+4)
/* #define IDEA32 */

#ifdef IDEA32
#define low16(x) ((x) & 0xFFFF)
typedef unsigned int uint16;
#else
#define low16(x) (x)
typedef word16 uint16;
#endif

typedef word16 IDEAkey[KEYLEN];

/* if using the gcc compiler, use the next define */
#define CONST __const__
/* if not using gcc compiler, use the following */
/* #define CONST   */


static void en_key_idea(word16 userkey[8],IDEAkey Z);
static void de_key_idea(IDEAkey Z, IDEAkey DK);
static void cipher_idea(word16 in[4],word16 out[4], CONST
IDEAkey Z);

/* define the multiply modulo 2**16+1.  This one has no
branches and
 * seems most appropriate for a superscalar processor
 */
#define MUL(x,y)  (x = low16(x-1), t16 = low16((y)-1), \
     t32 = (word32)x*t16+x+t16+1, x = low16(t32), \
     t16 = t32>>16, x = x-t16+(x<t16) )



/* idea.c */

#include "idea.h"


CONST static uint16 inv(uint16 x)
```

```
{
  uint16 t0, t1;
  uint16 q, y;

  if (x <= 1)
    return x;    /* 0 and 1 are self-inverse */
  t1 = 0x10001 / x;  /* Since x>=2, this fits into 16 bits */
  y = 0x10001 % x;
  if (y == 1)
    return low16(1-t1);
  t0 = 1;
  do
  {
    q = x/y;
    x = x % y;
    t0 += q*t1;
    if (x == 1)
      return t0;
    q = y/x;
    y = y % x;
    t1 += q*t0;
  } while (y != 1);
  return low16(1-t1);
}  /* inv */


/* Compute IDEA encryption subkeys Z */
static void en_key_idea(word16 *userkey, word16 *Z)
{
  int i, j;

  /* do shifts */
  for (j=0; j<8; j++)
    Z[j] = *userkey++;

  for (i=0; j<KEYLEN; j++)
  {
    i++;
    Z[i+7] = Z[i&7] << 9 | Z[i+1 & 7] >> 7;
    Z += i & 8;
    i &= 7;
  }
}   /* en_key_idea  */


/* Compute IDEA decryption subkeys DK from encryption subkeys Z */
static void de_key_idea(IDEAkey Z, IDEAkey DK)
{
  int j;
  uint16 t1, t2, t3;
  IDEAkey T;
  word16 *p = T + KEYLEN;

  t1 = inv(*Z++);
```

```
  t2 = -*Z++;
  t3 = -*Z++;
  *--p = inv(*Z++);
  *--p = t3;
  *--p = t2;
  *--p = t1;

  for (j = 1; j < ROUNDS; j++)
  {
    t1 = *Z++;
    *--p = *Z++;
    *--p = t1;

    t1 = inv(*Z++);
    t2 = -*Z++;
    t3 = -*Z++;
    *--p = inv(*Z++);
    *--p = t2;
    *--p = t3;
    *--p = t1;
  }

  t1 = *Z++;
  *--p = *Z++;
  *--p = t1;

  t1 = inv(*Z++);
  t2 = -*Z++;
  t3 = -*Z++;
  *--p = inv(*Z++);
  *--p = t3;
  *--p = t2;
  *--p = t1;

  /* Copy and destroy temp copy */
  for (j=0, p=T; j<KEYLEN; j++)
  {
    *DK++ = *p;
    *p++ = 0;
  }
}    /* de_key_idea  */


/*  IDEA encryption/decryption algorithm  */
/*  Note that in and out can be the same buffer  */
static void cipher_idea(word16 in[4], word16 out[4], CONST IDEAkey Z)
{
  uint16 x1, x2, x3, x4, t1, t2, t16;
  word32 t32;
  int r = ROUNDS;

  x1 = *in++;
  x2 = *in++;
  x3 = *in++;
```

```
  x4 = *in;

  do
  {
    MUL(x1,*Z++);
    x2 += *Z++;
    x3 += *Z++;
    MUL(x4, *Z++);

    t2 = x1^x3;
    MUL(t2, *Z++);
    t1 = t2 + (x2^x4);
    MUL(t1, *Z++);
    t2 = t1+t2;
    x1 ^= t1;
    x4 ^= t2;

    t2 ^= x2;
    x2 = x3^t1;
    x3 = t2;
  } while (--r);

  MUL(x1, *Z++);
  *out++ = x1;
  *out++ = x3 + *Z++;
  *out++ = x2 + *Z++;
  MUL(x4, *Z);
  *out = x4;
}  /* cipher_idea  */

/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --*/

/*
 * This is the number of Kbytes of test data to encrypt.
 * It defaults to 1 Mbyte.
 */
#ifndef KBYTES
#define KBYTES 1024
#endif

void main(void)
{
  int i, j, k;
  long l;
  IDEAkey Z, DK;
  word16 XX[4], TT[4], YY[4];
  word16 userkey[8];

  /* Make a sample user key for testing */
  for (i=0; i<8; i++)
    userkey[i] = i+1;

  /* Compute encryption subkeys from user key...*/
  en_key_idea(userkey, Z);
```

```
  printf("\nEncryption keys computed.");

  /* Compute decryption subkeys from encryption subkeys... */
  de_key_idea(Z, DK);
  printf("\nDecryption keys computed.");

  /* Make sample plaintext pattern for testing... */
  for (k=0; k<4; k++)
    XX[k] = k;

  printf("\n Encrypting %d KBytes (%ld blocks)...", KBYTES, KBYTES*64);

#ifdef TEST
  cipher_idea(XX,YY,Z);  /* encrypt plaintext XX, making YY */
  for (l = 1; l < 64*KBYTES; l++)
    cipher_idea (YY,YY,Z);    /* repeated encryption */
  cipher_idea (YY,TT,DK);     /* decrypt ciphertext YY, making TT */
  for(l = 1; l < 64*KBYTES; l++)
    cipher_idea(TT,TT,DK);    /* repeated decryption */
#endif  /* TEST */

  printf("\nX %6u   %6u   %6u   %6u \n", XX[0],XX[1],XX[2],XX[3]);
  printf("\nY %6u   %6u   %6u   %6u \n", YY[0],YY[1],YY[2],YY[3]);
  printf("\nT %6u   %6u   %6u   %6u \n", TT[0],TT[1],TT[2],TT[3]);

  /* decrypted TT should be the same as original XX */

}  /* main */
```

B.5.2 Assembly

The assembly for this program is quite long. To save paper, only the assembly for the sub-routine **cipher_idea** is given.

```
_cipher_idea:
      ;; Save the old frame pointer
      sw -4(r14),r30
      ;; Save the return address
      sw -8(r14),r31
      ;; Establish new frame pointer
      add r30,r0,r14
      ;; Adjust Stack Pointer
      addi r14,r14,#-64
      ;; Save Registers
      sw 0(r14),r3
```

```
        sw 4(r14),r4
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        sw 32(r14),r11
        sw 36(r14),r12
        sw 40(r14),r13
        sw 44(r14),r15
        sw 48(r14),r16
        lw r3,0(r30)
        lw r13,4(r30)
        lw r8,8(r30)
        addi r15,r0,#8
        lh r9,0(r3)
        add r3,r3,#2
        lh r12,0(r3)
        add r3,r3,#2
        lh r11,0(r3)
        lh r10,2(r3)
L33:
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r4,r9,r16
        lh r3,0(r8)
        add r3,r3,r16
        add r8,r8,#2
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        movi2fp f2,r4
        movi2fp f3,r3
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r4
        add r5,r5,r3
        add r3,r5,#1
        add r9,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r9,r4
        add r5,r0,r3
        sll r3,r9,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L36
        nop
        add r3,r5,#1
        add r5,r0,r3
```

```
L36:
        add r9,r0,r5
        lh r3,0(r8)
        add r3,r12,r3
        add r12,r0,r3
        add r8,r8,#2
        lh r3,0(r8)
        add r3,r11,r3
        add r11,r0,r3
        add r8,r8,#2
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r4,r10,r16
        lh r3,0(r8)
        add r3,r3,r16
        add r8,r8,#2
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        movi2fp f2,r4
        movi2fp f3,r3
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r4
        add r5,r5,r3
        add r3,r5,#1
        add r10,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r10,r4
        add r5,r0,r3
        sll r3,r10,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L37
        nop
        add r3,r5,#1
        add r5,r0,r3
L37:
        add r10,r0,r5
        xor r3,r9,r11
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r3,r3,r16
        lh r4,0(r8)
        add r4,r4,r16
        add r8,r8,#2
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
```

```
        movi2fp f2,r3
        movi2fp f3,r4
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r3
        add r5,r5,r4
        add r3,r5,#1
        add r7,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r7,r4
        add r5,r0,r3
        sll r3,r7,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L38
        nop
        add r3,r5,#1
        add r5,r0,r3
L38:
        add r7,r0,r5
        xor r3,r12,r10
        add r3,r7,r3
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r3,r3,r16
        lh r4,0(r8)
        add r4,r4,r16
        add r8,r8,#2
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        movi2fp f2,r3
        movi2fp f3,r4
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r3
        add r5,r5,r4
        add r3,r5,#1
        add r5,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r5,r4
        add r6,r0,r3
        sll r3,r5,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L39
        nop
```

```
        add r3,r6,#1
        add r6,r0,r3
L39:
        add r5,r0,r6
        add r3,r5,r7
        add r7,r0,r3
        xor r3,r9,r5
        add r9,r0,r3
        xor r3,r10,r7
        add r10,r0,r3
        xor r3,r7,r12
        xor r4,r11,r5
        add r12,r0,r4
        add r11,r0,r3
        add r15,r15,#-1
              ; (set (cc0)      r15)
        sne   r1,r15,r0
        bnez  r1,L33
        nop
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r4,r9,r16
        lh r3,0(r8)
        add r3,r3,r16
        add r8,r8,#2
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        movi2fp f2,r4
        movi2fp f3,r3
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r4
        add r3,r5,r3
        add r3,r3,#1
        add r9,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r9,r4
        add r5,r0,r3
        sll r3,r9,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L40
        nop
        add r3,r5,#1
        add r5,r0,r3
L40:
        add r9,r0,r5
        sh 0(r13),r9
        add r13,r13,#2
```

```
        lh r3,0(r8)
        add r3,r11,r3
        sh 0(r13),r3
        add r8,r8,#2
        add r13,r13,#2
        lh r3,0(r8)
        add r3,r12,r3
        sh 0(r13),r3
        add r13,r13,#2
        lhi r16,(65535>>16)&0xffff
        addui r16,r16,(65535&0xffff)
        add r4,r10,r16
        lh r3,2(r8)
        add r3,r3,r16
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sll r3,r3,#0x10
        srl r3,r3,#0x10
        movi2fp f2,r4
        movi2fp f3,r3
        mult f2,f2,f3
        movfp2i r5,f2
        add r5,r5,r4
        add r3,r5,r3
        add r3,r3,#1
        add r10,r0,r3
        srli r3,r3,#16
        add r4,r0,r3
        sub r3,r10,r4
        add r5,r0,r3
        sll r3,r10,#0x10
        srl r3,r3,#0x10
        sll r4,r4,#0x10
        srl r4,r4,#0x10
        sltu  r1,r3,r4
        beqz r1,L41
        nop
        add r3,r5,#1
        add r5,r0,r3
L41:
        add r10,r0,r5
        sh 0(r13),r10
        ;; Restore the saved registers
        lw r3,-64(r30)
        nop
        lw r4,-60(r30)
        nop
        lw r5,-56(r30)
        nop
        lw r6,-52(r30)
        nop
        lw r7,-48(r30)
        nop
        lw r8,-44(r30)
```

```
        nop
        lw r9,-40(r30)
        nop
        lw r10,-36(r30)
        nop
        lw r11,-32(r30)
        nop
        lw r12,-28(r30)
        nop
        lw r13,-24(r30)
        nop
        lw r15,-20(r30)
        nop
        lw r16,-16(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
        nop
        ;; Return
        jr r31
        nop
```

B.5.3 Reconfigurable Assembly

As with the previous section, only the assembly for the reconfigurable version of **cipher_idea()** is given.

```
_cipher_idea:
        ;; Save the old frame pointer
        sw -4(r14),r30
        ;; Save the return address
        sw -8(r14),r31
        ;; Establish new frame pointer
        add r30,r0,r14
        ;; Adjust Stack Pointer
        addi r14,r14,#-64
        ;; Save Registers
        sw 0(r14),r3
        sw 4(r14),r4
        sw 8(r14),r5
        sw 12(r14),r6
        sw 16(r14),r7
```

```
        sw 20(r14),r8
        sw 24(r14),r9
        sw 28(r14),r10
        sw 32(r14),r11
        sw 36(r14),r12
        sw 40(r14),r13
        sw 44(r14),r15
        sw 48(r14),r16
        lw r3,0(r30)
        lw r13,4(r30)
        lw r8,8(r30)
        addi r15,r0,#8
        lh r9,0(r3)
        add r3,r3,#2
        lh r12,0(r3)
        add r3,r3,#2
        lh r11,0(r3)
        lh r10,2(r3)
L33:
        lh r3,0(r8)
        lh r4,4(r8)
        lh r5,2(r8)
        lh r16,6(r8)
        ;; IDEA function 1
        fpga r9,r3,r9,#10
        fpga r11,r4,r11,#10
        fpga r10,r16,r10,#10
        fpga r12,r5,r12,#1034
        add r8,r8,#8
        xor r3,r9,r11
        lh r4,0(r8)
        ;; IDEA function 2
        fpga r7,r4,r3,#1035
        add r8,r8,#2
        xor r3,r12,r10
        lh r4,0(r8)
        ;; IDEA function 3
        fpga r5,r7,r3,#12
        fpga r0,r4,r0,#1036
        add r8,r8,#2
        add r3,r5,r7
        add r7,r0,r3
        xor r3,r9,r5
        add r9,r0,r3
        xor r3,r10,r7
        add r10,r0,r3
        xor r3,r7,r12
        xor r4,r11,r5
        add r12,r0,r4
        add r11,r0,r3
        add r15,r15,#-1
              ; (set (cc0)    r15)
        sne     r1,r15,r0
        bnez    r1,L33
```

```
        nop
        lh r3,0(r8)
        lh r4,2(r8)
        lh r16,4(r8)
        lh r5,6(r8)
        ;; IDEA function 1
        fpga r9,r3,r9,#10
        fpga r11,r4,r11,#10
        fpga r10,r5,r10,#10
        fpga r12,r16,12,#1034
        sh 0(r13),r9
        add r13,r13,#2
        sh 0(r13),r11
        add r13,r13,#2
        sh 0(r13),r12
        add r13,r13,#2
        sh 0(r13),r10
L41:
        add r10,r0,r5
        sh 0(r13),r10
        ;; Restore the saved registers
        lw r3,-64(r30)
        nop
        lw r4,-60(r30)
        nop
        lw r5,-56(r30)
        nop
        lw r6,-52(r30)
        nop
        lw r7,-48(r30)
        nop
        lw r8,-44(r30)
        nop
        lw r9,-40(r30)
        nop
        lw r10,-36(r30)
        nop
        lw r11,-32(r30)
        nop
        lw r12,-28(r30)
        nop
        lw r13,-24(r30)
        nop
        lw r15,-20(r30)
        nop
        lw r16,-16(r30)
        nop
        ;; Restore return address
        lw r31,-8(r30)
        nop
        ;; Restore stack pointer
        add r14,r0,r30
        ;; Restore frame pointer
        lw r30,-4(r30)
```

```
nop
;; Return
jr r31
nop
```

Bibliography

[1] Abbot, A.L. and others, "Finding Lines and Building Pyramids with Splash 2", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 155-163, 1994.

[2] Agarwal, L. and others, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 101-110, 1994.

[3] Arnold, J.M. and others, "Splash 2", 4th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 316-322, 1992.

[4] Arnold, J.M., "The Splash 2 Software Environment", IEEE Workshop of FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 88-93, 1993.

[5] Athanas, P. and others, "PRISM II: Compiler and Architecture", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 9-16, 1993.

[6] Athanas, P. M. and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis", IEEE Computer, Vol. 26, No. 3, pp. 11-18, March 1993.

[7] Betz, Vaughn and Rose, Johnathan, "Using Architectural 'Families' to Increase FPGA Speed and Density", ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 10-16, 1995.

[8] BRASS Research Group, http://www.cs.berkeley.edu/projects/brass/.

[9] Brown, Stephen, "FPGA Architectural Research: A Survey", IEEE Design & Test of Computers, Winter 1996, pp 9-15.

[10] Brown, Stephen, Muhammad Khellah, and Zvonko Vranesic, "Minimizing FPGA Interconnect Delays", IEEE Design & Test of Computers, Winter 1996, pp 16-23.

[11] Butts, M., "Future Directions of Dynamically Reprogrammable Systems", Proceedings of the IEEE 1995 Custom Integrated Circuits Conference, pp. 24.1.1-24.1.8.

[12] Capsi, Eylon and Nicholas Weaver, IDEA as a Benchmark for Reconfigurable Computing, 9 December 1996, available at http://www.cs.berkeley.edu/~eylon /cs252/index.html.

[13] Carrera, J.M. and others, "Architecture of a FPGA-based Coprocessor: The PAR-1", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 20-29, 1995.

[14] Galloway, David and others, "The Transmogrifier: The University of Toronto Field-Programmable System", Second Canadian Workshop on Field-Programmable Devices, Kinston, Ontario, June 1994. Available via ftp at ftp://ftp.csri.toronto.edu /csri-technical-reports/306.

[15] Dehon, André and others, "A First Generation DPGA Implementation", January 1995, http://www.ai.mit.edu/projects/transit/dpga-proto-fpd95.ps.Z.

[16] Dehon, André, "DPGA Utilization and Application", FPGA '96: 1996 ACM Fourth International Symposium on Field-Programmable Gate Arrays, pp. 115-121, February 1996. Extended version available at http://www.ai.mit.edu/projects /transit/transit-notes/tn128.ps.Z.

[17] DeHon, André, "DPGA-Coupled Microprocessors: Commondity ICs for the Early 21st Century", IEEE Workshop on FPGAs for Custom Computing Machines, Buell

and Pocek, eds., IEEE Computer Society Press, Los Alamitos, California, pp. 31-39, 1994.

[18] Drayer, T.H. and others, "A MOdular and Reprogrammable Real-time Processing Hardware, MORRPH", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 11-19, 1995.

[19] Electronics Research Laboratory, University of California, Berkeley, "Reconfigurable Processors", In a response to Broad Agency Announcement no. 96-07, March 19, 1996.

[20] Foulk, Patrick, "Reconfigurable Computing with SRAM Programmable Gate Arrays", More FPGAs, Moore and Luk, eds., Abingdon EE&CS Books, Oxford, England, pp. 70-81, 1993.

[21] Galloway, D., "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 136-144, 1995.

[22] Gokhale, M. and others, "SPLASH: A Reconfigurable Linear Logic Array", International Conference on Parallel Processing, pp. 526-532, 1990.

[23] Guccione, S.A. and M.J. Gonzalez, " Classification and Performance of Reconfigurable Architectures", Field-Programmable Logic and Applications, Moore and Luk, eds., Springer, pp. 439-448, 1995.

[24] Hauck, Scott and others, "The Chimaera Reconfigurable Functional Unit", IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 1997. Available at http://www.ece.nwu/~hauck/publications/Chimaera.pdf.

[25] Hauck, Scott, "The Roles of FPGAs in Reprogrammable Systems", submitted to Proceedings of IEEE, 1997. Availabe at http://www.ece.nwu/~hauck/publications /mFPGAhard.pdf.

[26] Hauser, J. R. and John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", IEEE Symposium on FPGAs for Custom Computing Machines, pages unknown, 1997, available at http://www.cs.berkeley.edu/projects /brass/garp.html.

[27] Hennessy, J. and D. Patterson, Computer Architecture a Quantitative Approach: 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[28] Hesener, A., "Cache logic: One FPGA Per Board", Electronic Engineering, pp. 91-96, March 1995.

[29] Hoang, D.T., "Searching Genetic Databases on Splash 2", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 185-191, 1993.

[30] Howard, N. and others, "FPGA Acceleration of Electronic Design Automation Tasks", More FPGAs, Moore and Luk, eds., Abingdon EE&CS Books, Oxford, England, pp. 337-344, 1993.

[31] Hunt, Doug, Advanced Performance Features of the 64-bit PA-8000. Available at http://www.hp.com/computing/framed/technology/micropro/pa-8000/docs/ advperf.html.

[32] Hutchings, B.L. and M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", Field Programmable Logic and Applications, Moore and Luk, eds., Springer, pp. 419-428, 1995.

[33] Iseli, C. and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor Using FPGAs", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and

Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 17-24, 1993.

[34] Iseli, C., and E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, pp. 173-179, 1995.

[35] Koch, A. and U. Golze, "A Universal Co-processor for Workstations", More FPGAs, Moore and Luk, eds., Abingdon EE&CS Books, Oxford, England, 1993, pp. 317-325.

[36] Kovac, Mario and N. Ranganathan, "JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard", Proceedings of the IEEE, Vol 83, No. 2, February 1995, pp. 247-258.

[37] Kumar, V. and others, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., 1994.

[38] Lewis, David and others, "The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System", FPGA '97: ACM Symposium on FPGAs, pp. 53-61, February 1997. Also available at http://www.eecg.toronto.edu/~jayar /research.tm2.tvlsi.ps.gz.

[39] MIPS Technologies, Inc., MIPS R10000 Microprocessor User's Manual, 1996. Available at http://www.sgi.com/MIPS/products/r10k/UMan_V1.1/ R10K_UM.cv.html.

[40] MIPS Technologies, Inc., R10000 Technical Brief. Available at http:// www.sgi.com/MIPS/products/r10k/r10000_Pr_Info/R10000_Tech_Br_cv.html.

[41] Moura, Cecile, SuperDLX: A Generic Superscalar Simulator, McGill University School of Computer Science, Montreal, Canada, 27 May 1993. Documentation,

simulator source code, and DLX compiler available at http://www-acaps.cs.mcgill.ca /superdlx.

[42] Pryor, D.V. and others, "Text Searching on Splash 2", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, 1993, pgs. 172-177.

[43] Rajamani, S. and P. Viswanath, "A Quantitative Analysis of Processor - Programmable Logic Interface", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, 1996.

[44] Razdan, R. and M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", MICRO-27, pp. 172-180, November 1994.

[45] Roelke, G.R. IV, A Framework for an Automated Compilation System for Reconfigurable Architectures, MS thesis, AFIT/GE/ENG/97M-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1997.

[46] Shanley, Tom, PowerPC System Architecture, Mindshare, Inc., Reading, Massachusetts, 1995.

[47] Schneier, Bruce, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley and Sons, Inc., New York, 1994.

[48] Slaney, Malcolm, An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank, Apple Computer Technical Report #35, 1993. Available at ftp://worldserver.com/pub/malcolm/Gammatone.math.

[49] Sun Microsystems, The UltraSPARC Processor: Technology White Paper. Online document available at http://www.sun.com/sparc/whitepapers/ UltraSPARCtechnology.

[50] Wang, Q. and G. Gulak, "An Array Architecture for Reconfigurable Datapaths", More FPGAs, Moore and Luk, eds., Abingdon EE&CS Books, Oxford, England, pp. 35-46, 1993.

[51] Wirthlin, M. J. and B. L. Hutchings, "A Dynamic Instruction Set Computer", IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, California, pp. 99-107, 1995.

[52] Wirthlin, M.J., et. al., "The Nano Processor: A Low Resource Reconfigurable Processor", IEEE Workshop on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Society Press, Los Alamitos, California, pp. 23-30, 1994.

[53] Wittig, R. and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", IEEE Symposium on FPGAs for Custom Computing Machines, Buell and Pocek, eds., IEEE Computer Societey Press, Los Alamitos, California, 1996.

[54] Xilinx Corporation, XC6200 Series v1.10, April 1996. Available at http://www.xilinx.com/partinfo/6200.pdf.

Vita

Christopher B. Mayer was born on January 3, 1969 in Athens, Georgia. He spent a majority of his childhood years in Bryan, Texas and graduated from Bryan High School in 1987. He then continued to college, earning a BS in Electrical Engineering at Texas A&M University in May 1992. He received an Air Force commission upon graduation and entered active duty in early 1993 and was assigned to Rome Laboratory, Rome, New York where he worked as a project engineer and contract manager. From there he went to the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a master's degree in Computer Engineering. After leaving AFIT, his next assignment is to the software analysis division of the Air Force Operational Test and Evaluation Center (AFOTEC) in the high desert of Kirtland AFB, New Mexico.

Permanent Address:

306 East Brookside Drive
Bryan, TX 77801

email: gcmayer@aol.com

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
A RECONFIGURABLE SUPERSCALAR ARCHITECTURE

5. FUNDING NUMBERS

6. AUTHOR(S)
Christopher B. Mayer, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB OH 45433-7126

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CGE/ENG/97D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NAIC/TACC
Keith Anthony
4180 Watson Way
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

The invention of the Field Programmable Gate Array (FPGA) has led to a number of interesting developments. One is the idea of providing custom hardware support for applications running on a computer. These reconfigurable computers have been shown to decrease the execution time for some applications. Based on past results, attention has subsequently turned to using reconfigurable computing in general-purpose computers (e.g. desktop and workstation environments).

This thesis develops a design for just such a computer. The design, FPGADLX, is based on a hypothetical superscalar computer running the DLX instruction set and is generic enough in principle to be adapted to any superscalar or VLIW processor on the market today. This thesis begins by examining FPGA technology and reviewing related reconfigurable computing efforts. Based on this review, requirements for a viable general-purpose reconfigurable computer system were developed. These requirements drove the development of the eventual FPGADLX design which covers hardware organization and operation, as well as modifications to the operating system and compiler. A software simulator which can emulate a significant portion of the design and run actual programs has been built.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Reconfigurable Computer, Reconfigurable Computing, Superscalar, FPGADLX, FPGA, Field Programmable Gate Array, Custom Computer, DLX Instruction Set, SuperDLX, Dynamic Execution	292
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL